MIT/LCS/TR-253

AN INTEGRATED APPROACH TO FORMATTED DOCUMENT PRODUCTION

Richard Ilson

This blank page was inserted to preserve pagination.

An Integrated Approach to Formatted Document Production

by

Richard Ilson

Massachusetts Institute of Technology

Laboratory for Computer Science

Cambridge Massachusetts 02139

This empty page was substituted for a blank page in the original document.

An Integrated Approach to Formatted Document Production

by

Richard Ilson

Submitted to the
Department of Electrical Engineering and Computer Science
on August 8, 1980 in partial fulfillment of the requirements
for the Degree of Master of Science

Abstract

Recent advances in printing technology have reduced the cost of typeset quality printers. Unfortunately, the production of attractively formatted documents requires typographic skill and special training on computer-based text processing systems. In response to this situation, we have developed the Etude text processing system. The principal characteristics of Etude are that it embodies substantial typographic expertise, and is based on concepts familiar to untrained users. Furthermore, Etude provides a real-time display facility that allows the results of editing and formatting operations to be seen immediately. Thus, Etude supports the entire process of producing decorously formatted documents.

Thesis Supervisor: Michael Hammer
Associate Professor of Computer Science

Keywords: Document Processing
Office Automation
Text Editing
Text Formatting

This empty page was substituted for a blank page in the original document.

Table of Contents

Table of Figures

This empty page was substituted for a blank page in the original document.

Chapter One

The Etude Text Processing System

Recent advances in printing technology have reduced the cost of typeset quality printers. Unfortunately, the production of attractively formatted documents requires typographic skill and special training on computer-based text processing systems. In response to this situation, we have developed the Etude text processing system. The principal characteristics of Etude are that it embodies substantial typographic expertise, and is based on concepts familiar to untrained users. Furthermore, Etude provides a real-time display facility that allows the results of editing and formatting operations to be seen immediately. Thus, Etude supports the entire process of producing decorously formatted documents.

An office is made up of persons with a variety of skills, many of whom are involved in producing documents. *Secretaries* are responsible for typing and changing documents according to the wishes of *professionals* in the office. Computer-based text processing systems, known as "word processing systems," assist the secretary in these activities by making it easier to type a document initially, and change it as necessary. In order to use word processing systems, a secretary must undergo a training period on the system, during which time he cannot be fully productive. The training process may be frequently repeated over time, because of the rapid turnover in the clerical labor market. The professional in the office, who is responsible for the content of documents, often cannot use a word processing system because he cannot invest the time to learn the system. The problem is compounded by the fact that professionals, unlike secretaries, would be intermittent users of a text processing system; a complex system, which requires steady and frequent use to

develop the expertise needed to use it effectively, is not of great value to the office professional. Today, therefore, professionals rarely use word processing systems. Because of an anticipated shortfall in the available population of office support staff, and because of the benefits that derive from direct use of advanced office systems, it is important that professionals be able to make effective direct use of automated office tools. By being simple to learn and easy to use, Etude addresses the needs of both office professionals and office support personnel.

By supporting the production of typeset quality documents, Etude goes beyond being merely an easy to use word processing system. Etude exploits the recent advances in printing technology, such as "dry" photo-typesetters, and electronic printers, using either dot matrix, ink jet, or laser technology. Electronic printers are characterized by their ability to print an unlimited variety of shapes at arbitrary positions on a page. The shapes printed are normally text characters in different type styles and sizes. In this respect, electronic printers are as capable as typesetting machines, although the output quality—in terms of resolution or sharpness—is somewhat lower. In addition, the shapes might be as complex as a drawing, logotype, or picture. Electronic printers are fast; the Xerox 9700 can print two pages a second. [7]

Although typesetting improves the appearance of a document, the advantage is more than aesthetic: a typeset document is easier to read and understand. It also requires less paper than its typewritten counterpart, thus reducing printing, copying, and mailing costs. Because electronic printers can print a (low resolution) typeset page, they share all these advantages of typesetting. Unfortunately, the advantages offered by these printers are accompanied by some drawbacks; complicated formatting instructions and aesthetic considerations are required to produce a typeset quality page. A system that allows a user without typographic skills to easily produce decorously formatted documents would hasten the acceptance of electronic

printers in offices. Etude, with its emphasis on integrating a text formatting facility into an easy to use system, directly addresses the issues and problems raised by the new printer technologies.

1.1 "Ease of Use" Problems and Etude's Solutions

In order to develop a system that is truly easy to use, it is first necessary to understand and identify the factors that make some text processing systems difficult to use. One problem is the low-level and detailed nature of the interface through which an operator communicates with the system. In many systems, the user must express himself in great detail, using terms that are unnatural to him and to his application. This is particularly pronounced in the case of formatting systems, which require the user to be an amateur typographer in order to produce a typeset quality document. The operator must invest a substantial amount of time and effort to manipulate the format commands of a document; the complexity of the interface to the formatter raises the cost of producing documents. Also, the average office worker does not usually possess the judgment and training needed to effectively use the capabilities provided by electronic printers. Providing these operators with a low-level command language with which to control such devices is a prescription for the production of ugly documents.

Both the editing and formatting functions of Etude allow a user to express himself in terms that are natural to him and his task. For editing, Etude has an "English-like" command structure, in which familiar primitives are combined to make commands. A typical command has the form: *action* / (optional) *modifier* / *object*. An *action* could be **delete** or **copy**; a *modifier* could be **next** or a number, like 3; and an *object* could be **word**, **line**, or **paragraph**. Thus, typical commands are: **delete 3 words** and **copy next paragraph**. The most common primitives are directly available

as special keys, and are logically grouped together.

The typical user of Etude does no direct formatting of his document, in the sense of providing low-level commands to the system selecting type faces, the spacing that is to be used, margins, and the like. Rather, the user merely identifies and names the components of the document in terms of the familiar structures of conventional office documents. For example, the operator might identify the document on which he was working as a *letter*, indicating within it the *return address*, the *address*, the *salutation*, the *body*, and so on; within the body, he would indicate the paragraphs and any other constituent structures that he needs. Etude utilizes a data base of formatting information to construct formatted documents in keeping with the user's specifications. Substantial formatting expertise is embedded in this data base, which can be modified or extended as needed, to conform with the requirements of individual offices and the people who work in them.

A second major problem is the delayed feedback loop implicit in contemporary "batch" formatting systems. In these systems, the operator inserts formatting commands in with the text of his document; after he concludes editing this text, he reprocesses it with a formatting system, and then prints the result. This delay considerably encumbers the process of constructing a document with a desired appearance. Also of major importance is what we call the "anxiety factor." Conventional computer-based systems of many kinds do not give the operator a feeling of security and control. Often a long time must elapse before the operator is sufficiently expert with the system to feel truly comfortable with it. Until then, the user feels as though he is walking a tightrope while wearing a blindfold. Because of the often obscure nature of the interface with which he is presented, he cannot fully anticipate the consequences of the actions that he performs. This leads to feelings of tension and uncertainty. Specifically, the user develops a fear of committing an unrecoverable error, and thereby becomes overly timid and cautious in his dealings

with the system.

Etude responds immediately to commands issued by its user. Display-based word processing systems achieve this goal by showing the user the results of his commands right away on the screen, making these word processing systems significantly easier to use than their non-display predecessors. Because Etude integrates editing and formatting into a single system—unlike word processing systems—it shows both the *content* and *appearance* of a document on its display. The operator sees and manipulates a fully formatted version of the document, one that represents a final image of what the document would look like if it was printed. This includes multiple type faces and sizes, variable leading (inter-line space), a variety of page layouts, and other capabilities typically associated with typeset documents. The purpose of providing this interactive, "real-time" formatting facility is to reduce the feedback loop, so that an operator specifying the appearance of his document will not have to wait until the output is produced by a printing device to determine if its appearance is suitable; he can see it right on his screen. If he is in the midst of an operation he does not wish to complete, the **cancel** key may be struck. An **undo** key is provided to enable the operator to roll back actions already completed, whenever an action yields an undesirable result. These facilities provide an operator with a secure feeling, encouraging experimentation and an incremental learning process.

A third significant problem is that even if a system is easy to use, it is often very difficult to learn. The learning process is frequently non-incremental: in order to perform any significant work, the operator must learn an extensive body of commands and features. Usually, it is not possible for the operator to develop a simplified model of the operation of the system, one which utilizes a smaller set of features. Instead, a full conceptualization and understanding of the system's capabilities is required, even to employ just a rudimentary set of commands.

Moreover, the transition from novice to experienced user is not a smooth one in most contemporary systems. On one hand, a system that makes extensive use of prompting, menus, and other such devices in order to simplify the use of the system for new users is likely to be extremely cumbersome for the same user once he has developed some familiarity with it. On the other hand, it is difficult for someone to become an expert on a system that is tailored to experienced users. In short, there is often a conflict between ease of learning and ease of use.

The user interface of the Etude system addresses this issue. Etude's working environment allows the user to control the amount of support he is given. Help information and menus are available should the operator need them; furthermore, they are optional, so an experienced user is not burdened with them. The operator can push the **help** key after any other key, and receive an explanation of the first key's function. He may also touch the **help** key whenever he is confused or uncertain; Etude then explains the current situation to the user, telling him how he got into the situation and what he may do next. When this explanatory material is displayed on the screen, the user's work does not disappear completely, so he will not fear he has "lost" anything. At any time, the operator can hit **menu** to see a list of the alternatives available to him at that time. Like **help**, these menus are context-dependent, showing only currently meaningful alternatives, but they lack the explanatory text that comes with **help**.

1.2 Software Architecture

A partial implementation of Etude,[1] which focused on exploring some of the

[1] "Etude" was chosen as the name for our first implementation effort because it was our practice exercise in building a rather complex text processing system. Etude is an Easy To Use Display Editor, for those who like acronyms.

more difficult implementation issues, was done using the CLU programming language [11], and forms the basis of the discussions in the remainder of this thesis. In this section we present a brief description of the software architecture of the entire Etude system.

The implementation of the Etude system is divided into two parts: the *user interface* and the *editor / formatter / display*. The *user interface* is responsible for parsing the keystrokes entered by the user, interpreting them, and then invoking the appropriate internal operations to realize the desired function. Most of the time a function of the *editor*, which is responsible for making changes to the document, is invoked. If the user's command does not involve changing the document, the *user interface* handles the function directly: this is the true for **help, menu,** and **cancel** functions. After the appropriate internal operations have been performed, the *user interface* updates the *session state*, which is a record of what actions have been performed, mainly for the purpose of implementing **help** and **undo**. The *formatter*, which reformats the regions in the document that have been changed (and appear on the screen), is then invoked. Finally, the *display* system is invoked, and it updates the screen image to reflect any changes made to the document. In the remainder of this section, we walk through several simple scenarios.

If the user types a text character, such as "i," when he is not in the midst of another command, the character is simply inserted into the document. The *user interface* instructs the *editor* to insert the character "i" into the document, and then updates the session state to indicate that the character was inserted. The *user interface* then invokes the *formatter*, which reformats at least the line the "i" was inserted into, and possibly more (if the line "overflowed"). Then the *user interface* asks the *display* system to update the screen. The *display* system redisplays at least the line containing the new "i"; again, more lines may be redisplayed if the insertion of the "i" causes changes on other lines in the document. After all these operations

are completed, the *user interface* waits for more keystrokes from the user.

All changes to the document follow this same basic pattern. A more complicated command, such as **delete 3 lines,** requires additional work from the *user interface* to parse the command, to invoke more general operations of the *editor,* and to record the operation in the session state. The operations of the *formatter* and *display* system remain essentially the same, although larger regions of text may need to be reformatted and redisplayed.

The *user interface* handles a **help** request by examining the session state, and constructing a temporary document containing the text of the help information. It allocates an area on the screen to display the text of the help information, then calls the *formatter* and *display* system on this temporary document, which results in the appearance of the help information on the screen.

1.3 An Overview of the Research Tasks

In this section, we present an overview of the work described in detail in the remainder of this thesis. This overview serves to motivate the remaining chapters, and gives the reader a context for understanding the work, as each component of the system is presented individually.

The focus of the work described in this thesis is the formatting of the text of documents, and the interrelationship between the formatting, editing, and display of a document. The design of the text formatter reflects the principal goal of the overall Etude system: to have it be simple to learn and easy to use. To achieve this goal, the formatter accepts *high level formatting commands,* and is *interactive*:

1. The user defines the format of his document by specifying the type of the document (e.g., "report" or "letter") and labeling portions of text (e.g., "quotation" or "chapter title"), rather than by specifying low-level

typographic commands. The formatter must interpret these specifications, and format the text appropriately.

2. As the user is editing and formatting his document, the Etude system continually displays a representation of how the document would look if it was printed. This requires the formatter to perform "incremental formatting," i.e., to be able to quickly reformat only portions of the document that both will be displayed on the screen and also need reformatting.

This thesis presents a solution to the problem of how to build and structure a system that meets these requirements.

A major problem to be solved in implementing the editor/formatter is defining the representation of the document. In addition to its content, a document in the Etude system has two additional aspects: its internal structure and its outward appearance; each of these aspects, described below, are included in the document representation.

- The *content* of the document is a linear sequence of text characters; it is set by the user as he creates and edits his document.

- The hierarchy that represents the *internal structure* of the document is built by the user as he creates and edits the document. At the root of the hierarchy is the document (e.g., report). Contained within are the document components (e.g., chapter, section, paragraph). At the bottom of the hierarchy are sentences and words.

- The hierarchy that represents the *outward appearance* of the document is built by the text formatter. The root of this hierarchy is the document, which is divided into pages, then columns, then lines.

Both the internal structure and outward appearance of a document are modeled by hierarchies. Each aspect forms a strict hierarchy taken by itself, but the two cannot be combined. For example, a *paragraph* (internal structure) might be completely contained within a *page* (outward appearance); a paragraph might

extend over two pages, with neither one containing the other; or a paragraph could extend over several pages, so that the paragraph would completely contain a page.

In using Etude, the user builds the internal structure of his document by identifying all the components, such as chapters, paragraphs, quotations, numbered lists, and italicized phrases. Most of the components of the document's internal structure have format specifications associated with them. Each such specification includes a number of format parameters, such as the type face, leading, and margins, that are appropriate for the document component.

However, the format parameters may only partially specify the formatting *environment* of a piece of text. The formatting environment is a total specification of all the typographic parameters in force at any point in the document. For those format specifications that do not completely specify the formatting environment, the desired value for the unspecified parameters is derived from other format specifications. For example, the size of type normally used for a "quotation" is slightly smaller than the type used for the body of the document; a format specification for the size of type for a quotation would specify the value *relative* to the type size of the containing text, rather than specifying an absolute type size. Thus, the actual size of type for a quotation would be derived from other components that contained the quotation.

The components of the outward appearance hierarchy are assembled automatically by the system, based on the format specifications associated with the document class and the internal structure components. The units that are assembled are called *boxes*. Representing the outward appearance of a document in terms of *boxes* was first done by Knuth in his text formatter TEX. Knuth describes how he assembles boxes to produce a formatted document:

> The main idea of TEX is to construct what I call *boxes*. A character of type by itself is a box, as is a solid black rectangle; and we use such "atoms" to construct more complex

> boxes analogous to "molecules," by forming horizontal or vertical lists of boxes. The final pages of text are boxes made out of lists of boxes made out of lists of boxes, and so on down to the individual characters and black rectangles, which are not decomposed further. . . The individual boxes of a horizontal list or a vertical list are separated by a special kind of elastic mortar that I call "glue." [10]

These concepts are modified and extended for the Etude system. For example, some boxes will need to contain special information so that they may be incrementally formatted and redisplayed.

As the user of Etude creates and edits his document, he modifies its content and the internal structure. The representation chosen for these two aspects must provide operations to insert and delete characters from the content, and to create and remove components of the internal structure.

As the user changes the document, the display must reflect the changes. The display system is responsible for keeping the screen's display consistent with the document as it is being edited. When redisplay is required, the user interface invokes the formatter on the text that will appear on the screen, then the newly formatted text is displayed. In order for the document to be reformatted and the screen to be redisplayed immediately after each change to the document, the amount of work (computation) done for each of these operations must be minimized.

When the document is changed, the nature of the change is analyzed with respect to what portion of the text needs reformatting, and the appropriate portion of text is marked unformatted. For example, if a character is inserted into the document, the line that the character was inserted in is no longer formatted. The formatter examines the portion of text it was asked to format—generally the text that will appear on the screen—and formats only the text in the region that is unformatted. As the formatting is done, the formatter marks the portions of text that have been

changed, to indicate to the display system what text needs to be redisplayed. The display system then examines the text, and displays all the text that was marked as changed.

The following list summarizes the capabilities of the formatter; all of the following are attributes of the formatting environment.

Left and Right Margins
: The left and right margins of the text to be formatted.

Line Layout
: Specifies how the formatter should format individual lines. It may flush the text against the left or right margin, or center the text between the margins.

Justification
: If justification is on, the formatter produces justified (straight) margins. If it is off, one margin (usually the right margin) is ragged.

Type Face
: Specifies the type face the text should appear in.

Leading
: The inter-line spacing of the lines of text.

Indention
: The amount of space that the first line of the document component should be indented. A positive number results in a *paragraph-style* indention, while a negative number results in a *hanging* indention.

Break
: How the first and last lines of the document component should be handled. The document component may require a line break both before and after it, only before, only after, or neither.

Above and Below
: The vertical space preceding the first line, or following the last line, of the document component.

Numbering
: A document component, such as a section, may be numbered. If it is numbered, the location and style (for example, arabic or roman) of the number are specified. (When a numbered document component is inserted or deleted, all document components whose numbers are changed are automatically updated.)

In order to format a region of text, the formatting environment for the text must

be derived. The internal structure hierarchy is used to obtain this formatting environment. This is done by retrieving the format specifications associated with all the document components that contain the text to be formatted, and using an inheritance scheme to derive the particular local formatting environment from this set of specifications.

1.4 Thesis Organization

In Chapter 2, we present a survey of existing text processing systems, and describe how they relate to the Etude system. Chapter 3 presents our model of the structure of documents, as used by the Etude system. It addresses the issue of integrating the content, internal structure, and outward appearance of a document into a single, easily modifiable structure. Chapter 4 describes the attributes that govern the way Etude formats a document, and also explains how Etude determines the formatting environment for a region of text in the document. In Chapter 5, we tell how the text formatting and display systems work, emphasizing their *incremental* reformatting and redisplay capabilities. In Chapter 6 the automatic numbering system of Etude is described. Lastly, in Chapter 7, we evaluate the implementation of Etude, and suggest avenues for improvement and extension.

Chapter Two

Survey of Related Work

In this chapter we give an account of several sophisticated text processing systems in use today. Computer-based text editing and formatting systems have been used at M.I.T. for years, and we describe some of these systems. It is important to note that these systems were generally designed to be used by computer professionals, rather than by office personnel, who will be using Etude. Thus, it is also necessary to examine commercial text processing systems used in offices ("word processing systems"). We also look at composing systems used by typographers.

Generally, the designers of advanced text processing systems have either concentrated on providing a high degree of functionality in their systems, or focused on making their system easy to use. For each system described, we focus on the aspect of the system—either the user interface or the functionality—that is novel. The remainder of this chapter contains descriptions of the following systems:

Emacs	A display editor with a high degree of functionality.
Doc	A display editor with a good user interface.
Wang	A commercial word processing system that is easy to learn.
Scribe	A text formatter that is easy to learn and use.
TEX	A text formatter that produces high typographic quality documents.
Atex	A commercial text processing system for typeset documents.
Bravo	An integrated, interactive display text editor and formatter.

2.1 Emacs

Emacs is a real-time editor primarily intended for display terminals. [23] Emacs is "extensible," which means that users can add new functions to the editor. Unfortunately, the language that extensions are written in is not easy to learn, so that only relatively sophisticated users write extensions. Fortunately, Emacs was developed in an environment where there are many sophisticated users, and Emacs now contains a large number of useful functions. Emacs has commands that are specifically for editing of computer programs; these commands will not be detailed here.

2.1.1 Functionality

Basic cursor positioning commands allow moving the cursor: forward or backward over characters, words or lines; to the beginning or end of lines, sentences, paragraphs, pages, or the whole document; to the next or previous screen of text.

To insert text, the characters are simply typed in. A "quote character" allows insertion of text that might otherwise be recognized as a command to Emacs. Commands that are used frequently are invoked by typing one or two characters, preceded by one of several command keys.

A "region" of text is the set of characters between a "mark" and the current cursor position. Marks are saved on a stack. Operations involving marks include: mark beginning or end of buffer; mark current position; pop mark, and optionally move cursor there; exchange cursor and mark; mark word, paragraph, or page. There is no indication on the screen of where the marks are.

There are commands to delete or kill text. "Killed" text is saved, and may later be unkilled (i.e., brought back into the document). Available operations are: delete forward or backward characters; kill words, lines, or region; insert (unkill) the *n*th most recent string killed; "kill" region without actually deleting it—used in

anticipation of unkilling it later, to copy the text.

Commands are implemented to exchange the pair of characters or words before the cursor. Also, all the characters in a word or region may be made uppercase or lowercase, or just the first character of each word may be capitalized.

Emacs uses the idea of an "incremental search," in either the forward or reverse directions. As characters are typed in, the accumulated string is searched for, and the cursor is positioned at the point in the buffer that matches the string typed so far. One may delete a character in the search string, and the cursor is repositioned appropriately. It is easy to repeat the search with the same string in either the forward or reverse directions.

To search and replace strings within the buffer, a general "Query Replace" function may be invoked; the user would then be asked to confirm each replacement. The user confirmation may be disabled.

Emacs allows the user to create named buffers; each buffer may contain a different document. The commands that operate on buffers are: list buffers, which will list the name, document file, and mode (see below) of all the buffers; append region to specified buffer; select or create a buffer, given its name; kill specified buffer; list all the buffers, and ask if each should be killed.

Emacs supports two windows; while working within a window, any buffer may be selected. Typeout inside one window will stay there until the user edits in that window—thus, information (another buffer's contents or the result of a "help" request) may be left on the screen while editing in another window. Commands in Emacs that manipulate windows are: display one window; display two windows; switch to other window (if both windows are showing, pointer merely moves to other window—otherwise, the other window is displayed); grow or shrink current window, changing the number of lines it uses.

Associated with each buffer is a mode. One would normally specify "Text Mode" to edit English text. "Auto Fill Mode" will automatically break lines of text on input at any desired column ("word wrap"). "Word Abbreviation Mode" allows the user to abbreviate text with a single "word," and have Emacs expand the abbreviation automatically as soon as the abbreviation has been typed in. There are also modes for editing tables and indented text.

Numerical arguments may precede most of the above commands. The argument is usually interpreted as a repetition-count.

Emacs is integrated with some other systems on the computer. While in Emacs, one can edit a directory, use a mail subsystem, or access a general information program.

Although Emacs does not format text, several Emacs commands generate control sequences various text formatting programs.

2.1.2 User Interface

Emacs can be used from a standard CRT computer terminal with a standard keyboard, although it is more easily used with a special keyboard (one with a "Meta" key, which works like an additional "Control" key). Since all text characters are self-inserting, Emacs uses the control key and the escape (or meta) key to distinguish commands from text. Thus, the user is required to remember and repeat control and escape sequences in order to perform basic manipulations.

The less frequently invoked commands that do not have their own short key sequence, or those that require string arguments, are invoked by using the "minibuffer." Use of the minibuffer is requested, and a small window appears at the top of the screen. In this window, one may type and edit commands, using the command's long name. When done, the appropriate escape sequence is typed, and

the command is then executed. Any number of commands may be entered in the minibuffer.

Emacs has a number of self-documentation facilities. One may list all commands; however, this list would probably be too long to be of much use. Instead, one may list only a subset of these commands by specifying a string of characters. Only those commands for which a partial match exists will be listed. For example, specifying the character string "paragraph" would list all the commands for manipulating paragraphs. When more complete information is desired, one may request a full description of any command.

2.2 DOC

DOC is a text editor developed by V. R. Pratt at M.I.T. [17]

> The main difference between DOC and EMACS is the way in which each achieves coverage of the large range of operations it offers. The EMACS philosophy is to offer a very large set of one-character commands to cover what is wanted most frequently, with two-character commands being reserved for less frequently used facilities. Such an approach . . . has the drawback of burdening the user with a large and not particularly mnemonic vocabulary. In contrast, the DOC philosophy is to have a small and highly mnemonic vocabulary, and to achieve its large range of commands by permitting the basic commands to be used in combination.

DOC has a command vocabulary of about 30 commands, each associated with an English word. The commands are "English-like" not only from the standpoint of being mnemonic: they are also used in the same way English words are used. Specifically, they are combined, according to grammatical rules, to form phrases. The resultant phrase is a command to DOC. Thus, one types in edit actions in the same way one would normally "say" them; for example, "back 3 words," or "erase 2 lines."

There are five classes of commands: nouns, verbs, adjectives, imperatives, and golfball (motion). The nouns, verbs, and adjectives are used in conjunction with one another. The nouns (or objects) include: character, word, line, paragraph, file, text (used to search for a text string), and object (the text the cursor was just moved over). The verbs are: erase, dump (for making a copy of something to be moved elsewhere), underline, justify, and case-lower/upper (for changing case). The following adjectives modify the noun they are used with in any command: any number (1, 2, 3, etc.), back, right-end-of, and whole.

The imperatives and golfball commands are generally used by themselves. The imperatives are undo (for undoing the last operation), get (for getting back what has been "dumped"), make (a search and replace operation), and visit (for file manipulations). The golfball commands allow the user to move the cursor left, right, up, down, and for retracing the history of the cursor's recent movement.

2.3 Wang

The Wang Word Processing System is one of the more successful commercial word processing systems. [22] One of its main advantages is that it is quite easy to learn to use. It accomplishes this through the use of "menus," which list the various operations the user may perform. Menus are hierarchically structured into groups of related functions. There is a basic menu, which appears when the system starts up. It lets the user create, edit, or print a document, or choose another function. These other functions include: special print functions, document index, document filing, telecommunications, and others. Each of these functions have their own menus associated with them.

In the normal mode of operation, as a character is typed, it replaces the character at the current cursor position. This differs from the normal operating mode of the

computer-based editors, such as Emacs and Doc. However, both Emacs and Doc offer this mode of operation as an option. In the Wang system, one enters "insert mode" when characters are to be added to the text.

When insert mode is selected, the text from the cursor to the end of the screen is removed from the screen. The operator types the material to be inserted (which is highlighted), then touches "execute." The system then automatically realigns the rest of the document. With this scheme, all text is effectively inserted at the end of the screen, so no problems with reformatting the screen on text input are encountered.

Whenever the operator initiates an action where the system requires additional input (text or commands), a highlighted prompt appears in the upper right hand corner of the screen, notifying the operator what the system expects next. Novice or occasional users find this useful, and seasoned operators can easily ignore the prompt.

There is a large "cancel" key, far away from the main keyboard, which allows the operator to cancel a command at any time, and a large "execute" key, which is used to confirm some commands.

2.4 Scribe

Scribe is a "batch" text formatter, written by Brian Reid at Carnegie-Mellon University. [18]

> The guiding principle that shaped every aspect of the design of Scribe is that most people who produce documents don't know or care about the details of the formatting involved. To this end, those details are determined by information in Scribe's database and not by commands from the user...
>
> The Scribe system was designed to make document production easy for the non-expert,

> and to allow him to make small changes to the formats and styles without needing to learn much about how the program works. Scribe is not a programming language.
>
> Scribe does not have "commands" in the usual sense of the word: its commands are not procedural...

Scribe was initially designed to format the types of documents produced at a university, but is not limited to university documents. Scribe takes as input a file of text, with embedded format commands, and can produce output for a number of different devices. Scribe will do the formatting differently for the different devices, producing the best representation of the "final" copy that it can for any particular device.

Given that Scribe is not interactive, it does try to be flexible. For example, when delimiters are called for in a command, it will accept anything a user might imagine to be a delimiter, such as brackets, parentheses, or quote marks. The scope of a command may be specified by the delimiters above, or by "begin command" and "end command," if more appropriate.

The commands of Scribe may be grouped in a hierarchical fashion. At the highest level are those commands that specify the document type and its style. Currently, Scribe knows about: reports, which have a title page, numbered chapters, sections, subsections, and a table of contents; manuals, which are like reports with an index; articles, which are like reports without chapters; letters, with or without a letterhead; posters, a single-page poster or announcement; and slides for an overhead projector, where font and line spacing have been chosen to maximize readability. Within these classifications, one can request different styles. A different style manual might use a different set of type fonts, or might have different margins and line spacing. Note that at this level the user does not specify what font or line spacing he wants, but just asks for "Manual, Form 2."

At a level down from the document type are commands to specify titles, headings,

and sections. These commands are in two groups: one for a document without a table of contents, and another set for documents with a table of contents. The heading commands for a document lacking a table of contents are different, because there is no need for Scribe to give the heading a number or assemble the table of contents. These commands are: majorheading, to get large letters, centered; heading, to get medium-sized letters, centered; and subheading, to get normal-size boldface letters, flush to the left margin. In document types with a table of contents, Scribe gives the following sectioning commands: chapter, section, subsection, and paragraph; appendix and appendixsection; unnumbered; and prefacesection.

Scribe can generate page headings and footings. Each is divided in three parts: left, right, and center. Different headers and footers may be specified for the even and odd pages. Normally, a command to change the header or footer will take effect on the following page; this may be changed by requesting the header or footer of the current page to be changed.

Scribe has commands for generating a title page of a university report or manual. These are: titlepage; titlebox, which positions the text in a box that will be reproduced on the cover of the report; copyrightnotice; and researchcredit, for explaining where one gets one's money.

Down one more level, beneath sectioning commands, are the commands that allow one to insert various things in the middle of running text. These "insert" commands include: quotation, a text quotation (excerpt); verse, which will start a new line for each line in the verse (and will format appropriately if lines are too long); example, an example of computer type-in or type-out, which will appear in a type face that is designed to look like computer output; display, which is like example, except the normal body type face is used; center, where each line is centered; verbatim, where characters are copied exactly, without formatting (a fixed-width font is used); format, like verbatim, except a variable width font is used,

and normally the user would use special tabbing and formatting commands (described later); itemize, which indents paragraphs and places a tick mark before each; enumerate, which indents paragraphs and numbers them; description, which places a keyword or phrase flush left, then indents the remaining paragraph; equation, which is like display, except an equation number is generated and placed flush right; and theorem, which is like quotation, except it heads the text with "Theorem," followed by the theorem number.

Another type of insert is a "figure." Figures have three parts: a figure body, a caption, and a figure number. Figure bodies may be produced in any way. One may use the standard Scribe insert commands, such as format, verbatim, or example. Other ways are: blankspace, which will leave a blank space of a specified size (Scribe understands various units of length); picture, which will put a picture in the document if there is an image picture file; fullpagefigure, for a figure that requires an entire page; and blankpage, to generate a page with no text, only the header and footer.

Scribe provides several ways to get text or numbers formatted into columns. For simple formats and small tables, it is suggested that one use the verbatim command (which will output exactly what was typed in, in a fixed width font). Using verbatim, one would format "by hand." For more complex formatting, Scribe has a tab stop mechanism. Commands include: tabclear, to clear all tabs; tabs, to set tabs at specified horizontal positions; tabdivide, which will divide the formatting area into a particular number of columns; and the command "\", which tabs to the next tab stop. Other capabilities are: overprinting; a "return marker," to mark a horizontal position on a page; and the ability to center or flush right text with respect to tabs stops or the right margin.

Lastly, on this level, are commands to control word, line, and page breaks. Scribe will break lines between words, at blanks. One may make a blank significant, so that

Scribe will not break at that blank, by using a command. Another command will make all blanks in a delimited region significant. There is a command to allow Scribe to break where it normally would not, and to force a line break at a particular place. Scribe does not do automatic hyphenation, nor can it ask the user to hyphenate a word when it would be desirable.

If Scribe is processing an insert when a page fills up, it will just break the insert, and continue it on the following page. To prevent this, one may specify "float" when defining the start of an insert. In this case, if the insert would not fit on the current page, the text will continue without interruption, and the insert will appear at the top of the next page. To require that a new page be started if an insert doesn't fit on the current page, "group" may be specified at the beginning of the insert. Inside a grouped insert, a "hinge" command will allow the insert to be broken at that point.

At the lowest level are the commands to change fonts, get special characters, and generally change the "style" of a document. In a preface to this section Reid states: "Although Scribe's basic approach to document production is to provide its users with a large menu of document types and discourage them from tinkering with details, we recognize that the urge to tinker is incurable."

Within the current font, there are commands for: italics; underline non-blank characters; underline all characters; underline alpha-numerics, but no punctuation or spaces; boldface; roman (the normal type face); typewriter font; super- and sub-scripting; small capitals; greek characters; overbar; and bold italics. If one wants a different font (meaning a set of 10 to 15 type faces and sizes), the "font" command is used.

The "style" command changes the setting of certain of Scribe's internal parameters. The database entry for each document type provides values for all these

parameters; thus, the style command would be used to override some of these values. The style parameters are: indentation, to set the amount of indention of the first line of a paragraph; spacing, to set inter-line spacing; spread, to control the spacing between paragraphs; justification, either on or off; leftmargin, rightmargin, topmargin, and bottommargin, to set margin sizes; paperlength and paperwidth; and footnotes, to control the way footnotes are numbered.

Scribe does automatic bookkeeping of cross references of various kinds. Scribe lets the user create a label to mark a point in the text. The section number of the place may be referenced with the "ref" command, while the page number may be referenced with the "pageref" command. The "tag" command lets one reference numbers of things Scribe has automatically numbered. For example, one may tag an entry in an "enumerated" list, or a figure number. The "ref" command could then be used to reference the list or figure number.

The "footnote" command will automatically generate a footnote number and place the footnote at the bottom of the page.[2] The "index" command will place a word in the index that Scribe will generate automatically.

Scribe has sophisticated facilities for dealing with bibliographies and citations. "A bibliography is a labeled list of books, articles, papers, and the like. A citation is a marker in the text that refers to an entry in the bibliography. . . . The Scribe bibliography facility does three things: 1) Selects from a database the bibliographic entries that are actually cited; 2) Formats them into a bibliography and assigns a number or label to each; and 3) causes the correct numbers to be placed in the citations in the text." Scribe will format both the bibliography and citations in the

---

[2]Reid states: "In providing such a simple footnote mechanism, we feel a responsibility to advise you to use it sparingly. Footnotes seriously interfere with the readability of a paragraph, and their excessive use will distract the reader rather than help him." Even in a relatively constrained formatting system, Reid recognizes there are still ways people can produce "bad" documents.

"proper" way, given the name of the journal.

Scribe has facilities to aid production of large documents. The large-document facility includes:

> An include command, that lets you compose a large document from any number of separate files, each one of which is of manageable size.
>
> A part command that lets you process a component file independently of the whole document, yet still have page numbers, section numbers, chapter numbers, and cross references come out right.
>
> A use command to request that Scribe use some private or custom edition of its database.
>
> An outline of your document, automatically generated by Scribe in a separate file, showing the sectioning structure of your document and its cross-reference points, to help you manage its organization.
>
> A word counter and vocabulary analyzer.

Also, one may use the "value" command to access internal strings. For example, one may access the date (in a choice of styles), month, year, weekday, time, name of the manuscript, and title of the current section.

2.5 TEX

2.5.1 Overview

Knuth's documentation on his system for technical text, "TEX," explains both the typographic and programming issues related to text formatting. [10] Although a major feature of the system is its ability to handle the formatting of mathematical text, it contains a complete discussion of its normal text formatting procedure. Unlike other formatters, TEX's objective is to produce documents "whose typographic quality is comparable to that of the world's finest printers." Note that TEX

is not an integrated document production system; its input is a prepared file containing text, formulas, and appropriate formatting commands.

TEX handles the extended character set of a typical type font (for example, opening and closing quote marks, different types of dashes and spaces). It also automatically recognizes places to insert ligatures and do kerning, advanced typographic features not found in most other computer text formatting systems. TEX automatically positions many different types of accent marks correctly over (or under) characters.

TEX commands are part of the text file, preceded by an "escape character." The system has commands to specify type fonts and sizes. One may "group" commands, so that, for example a font could be changed for just a phrase within a group, without affecting the font specification outside the group. Groups may be nested arbitrarily deep; this allows rather complex formatting instructions to be built up. TEX allows definitions (macros) with arguments to be declared, so that frequently used control sequences may be referred to easily.

2.5.2 Functionality and Internals

TEX makes up pages by pasting together *boxes* with *glue*. Boxes are rectangular objects with three associated measurements: *height, width,* and *depth*. These are measured with respect to a *base line,* and a *reference point* at the left of the base line. A single character of a font is a (simple) box. While forming lines, TEX will normally line up the base lines of the boxes; however, it can move the reference points up or down to do super- or subscripting. There are also *black boxes,* completely filled with ink, for making horizontal or vertical rules. Everything on a page is made up of these boxes, pasted together.

To paste these boxes together, TEX uses *glue*. Glue has three attributes: its

natural *space*, its ability to *stretch*, and its ability to *shrink*. When making larger boxes, such as lines or pages, TEX will shrink or stretch each piece of glue in proportion to its stated shrinkability or stretchability. After punctuation, the stretchability of the glue increases (and the shrinkability decreases), to allow for more space after punctuation. By making the stretchability of the glue at one or both ends of a line "infinite," TEX can produce lines that are flush left, flush right, or centered.

One of the novel features of TEX is the way it breaks paragraphs into lines. TEX waits until it reaches the end of a paragraph, and then determines the best way to break the paragraph into lines. Knuth claims that this approach to the problem "requires only a little more computation than the traditional methods, and leads to significantly fewer cases when words need to be hyphenated." TEX combines this paragraph breakup scheme with an extremely cautious approach to hyphenation. One can specify that TEX should "try harder" if the initial result is unsatisfactory; also, one can indicate that it is undesirable to break in a particular place. Knuth details his paragraph breakup and hyphenation algorithms in the TEX manual.

TEX groups things into pages in much the same way as it makes up paragraphs, except for the lookahead feature. Each page break is made once and for all when the "best" place is found. Again, Knuth details his page breakup algorithm, including: calculation of inter-line glue, inserting illustrations, and final page makeup (for example, appending page numbers).

In addition to features previously mentioned, TEX has facilities to: insert "leaders" (either horizontal or vertical); set, increment, and insert counters; set up hanging indents; process "conditional text"; set up an "output" routine, specifying what the final page format should be.

2.5.3 User Interface

As currently implemented, TEX is a computer program running on a general purpose time-shared system. Although one may input directly to the system, generally one passes to TEX a file containing text interspersed with formatting commands.

A user may tailor TEX to his desire by redefining recognized control characters and by adding macro definitions. The basic format contains mostly definitions of mathematical functions, but there are some general functions defined that allow a user to do most of the basic formatting functions.

TEX accepts dimension specifications using various units of measure, including points, picas, inches, and metric units. Thus, both novices and experienced printers alike may feel comfortable using TEX.

2.6 Atex

The Atex system is a complete video terminal-oriented composition and editing system. [21] The system comes in two varieties: a commercial version and a newspaper version. This section will briefly list novel or interesting aspects of the commercial system.

2.6.1 System Features

Atex's video terminal is just an extension of the host computer. Specifically, the screen image is taken directly from the main memory of the host. The screen displays 25 lines of 80 characters. There are normally 22 lines of text, with 3 lines of job header. The user may move his cursor between the two areas, and the system will remember the previous cursor position in the other area.

Characters may be displayed normal or bold, with reverse video and/or underlined. Characters are stored using a full sixteen bit word, so that the display mode information is part of the character. "Format files" define the typographic appearance of a job. Each type face defined in the format file would probably be displayed in a different display mode. Thus, there are no type face change commands imbedded in the text, and one can change the type face associated with any particular display mode by just changing the definition in the format file.

The terminal keyboard has a lot of keys: a main keyboard (a duplicate of the IBM selectric keyboard with "a beautiful typing touch"); cursor control keys; a bank of editing keys; a bank for typesetting commands; a bank for the various display modes and system commands; and a row of buffer keys.

All operations on the system can be performed interactively, up until the final outputting to a CRT or phototypesetter. Batch processing is also supported.

The Atex system supports basic display editing functions. You may "define" a text element, then "act" on it. You may "save" the defined text, and that text is then associated with one of the sixteen buffer keys.

Atex uses a total dictionary approach for hyphenation. Each word to be hyphenated is looked up in a 110,000 word dictionary. Any word not in the dictionary is hyphenated by an algorithm, and is also flagged as such.

Formatting features include: specification of minimum, maximum, and optimum interword spacing; a hyphenation-justification program that is output device independent; kerning, ligatures, and accents are supported; a program is available for arranging tabular display on the screen; some mathematics formatting capability; a page-makeup program.

2.7 Bravo

Bravo is an interactive editor and formatter designed by Butler Lampson and Charles Simonyi at Xerox's Palo Alto Research Center. [24] It runs on an Alto computer system with a high-resolution CRT divided into a number of areas: a major area where the text is displayed and edited; a line containing the editor "state" (last text inserted, deleted, searched for); an area containing the most recent commands; and a small area at the left of the screen scrolling through the document.

As text is entered by the operator, it appears on the screen in the correct font. Bravo can right-justify text as it is typed in. As the line fills up, the inter-word spacing is decreased, until no more words can fit on the line. The word that overflowed the line is moved to the next line and the inter-word spacing is expanded to fill the line.

Etude makes use of a "mouse" to edit text. A mouse is small object with three buttons attached to the keyboard by a thin wire (its tail). As the user moves the mouse in any direction on the desk, a cursor on the screen follows the motion. To edit the text, the appropriate portion of text is delimited. To refer to a character, the mouse is moved so the cursor is positioned on the character, and a button on the mouse is pushed. If a word is to be delimited, another button is pushed. If an area of text is to be defined, the mouse is moved to the final character of the area, while another button is depressed.

Bravo has two modes: *insert* and *alter*. In insert mode, characters appear on the screen at the cursor position as they are typed. In alter mode, each alphabetic key specifies a particular editing (or formatting) operation. There is an "undo" operation, which reverses the effects of the most recent operation. Multiple windows into the same document or different documents are supported by Bravo.

Formatting commands are specified in a manner similar to the editing commands.

The area of text that is to have a special format is defined, and the formatting information, such as type face, centering, leading, or margins, is entered. The text then appears on the screen in the new format.

When one has finished editing and formatting the document, it is sent to the appropriate hardcopy device. Unfortunately, what you see is not what you get; because of the limited resolution of the Alto display, Bravo does not usually display the columns of text as they it will look when output. In particular, the line breaks displayed will differ from those on a printed copy of document. There is a "hardcopy" mode which does show individual columns of text exactly as they will appear when printed. This is useful for checking that lines have been broken at appropriate points.

Bravo lacks some important features. It does not support interactive or automatic hyphenation. It will support multi-column documents, and paginate a document either automatically or interactively. However, it is difficult to define a complex page layout. And, it will not display a complete page on the screen. Thus, the user is required to print the document (or use a special page display program) in order to see how the page will look. For these reasons, Bravo is not significantly easier to use than the non-interactive formatters previously described.

2.8 Conclusions

In all the systems surveyed, those features that are appropriate for an office text processing system were isolated and integrated into Etude's design.

The Wang system demonstrates that menus and system prompts make a system easy to learn. These facilities, when properly designed, do not encumber an experienced user: a proficient Wang operator will ignore the prompts and quickly

step through the menus when he knows exactly what he is doing.

The "natural language" approach to text editing, similar to DOC's, is a straightforward, natural way to achieve a large coverage of editing commands, while requiring little memorization on the user's part.

Scribe's approach to document formatting is particularly appropriate for the office environment. It allows users to specify formats in familiar terms, and it automatically does a variety of normally tedious bookkeeping tasks.

Emacs shows that a highly functional editor assists the editing process for experienced users. Many of Emacs's features help a person in the computer program writing process. Advanced editing features appropriate for office documents, such as a table formatting and editing system, would be an analogous features appropriate for the Etude system.

The TEX system proves a computer formatting program is capable of producing typographically beautiful documents. Its underlying document representation and associated formatting algorithms are simple, practical, and elegant. With suitable enhancement, much of TEX's internals can be used in the Etude system.

The Atex system is a successful commercial text processing system. It provides a good text editing facility, and produces typeset output. It has many of the features required of a "polished" commercial system.

Bravo was the first system to provide an interactive environment for both editing and formatting documents. With Bravo, there are no formatting commands interspersed with the text. Rather, the system shows the results of the formatting commands by displaying formatted text on the screen.

Chapter Three

A Model of the Structure of Documents

This chapter is devoted to presenting Etude's model of documents. We first examine how existing text processing systems model documents, and explain why these simpler models are not sufficient to serve as the model for documents manipulated by the Etude system. From this analysis we will assemble a model of the structure of documents; the model will include the aspects of documents that are relevant to the editing and formatting functions of Etude. After the model is defined, we will describe an implementation of the model that will be used to represent all documents in the Etude system.

There are currently two kinds of text processing systems in popular use: text *editors* and text *formatters*. People use text editors to create and modify the content of documents, and they use text formatters to compose the outward appearance of documents. The text editors and text formatters mentioned in the following discussion have been described in Chapter 2.

A typical text editor, such as Emacs or the Wang system, has a model of a document that includes only the document's content—the sequence of text characters. The model does has information about the outward appearance of the document as it appears on the screen of the text editor; for example, the displayed text is broken into lines. The outward appearance as modeled by the text editor (and visible on the screen), however, bears no relationship to how the document would look if typeset. (There may be formatting commands intermixed with the content, but text editor can only treat them as ordinary text.) The text editor's model, therefore, includes no direct information about the document's ultimate

outward appearance.

A text formatter, such as Scribe or TEX, has a more elaborate model of a document. Its model includes information about the outward appearance of a document. There is information about how the document is broken into pages, and where every character of text is located on each page. The text formatter builds its model from the text editor's model by interpreting the formatting commands included in the content. The translation required to create a formatted document is time consuming, and is generally done infrequently, in a "batch mode." Once the model is created, it cannot be directly modified, because no provisions are made to allow the user to edit the text formatter's model directly. If the user wishes to change something, he must use the text editor to make the change, and then invoke the text formatter, which builds a new model.

The Etude system differs from conventional text processing systems because it operates on both the content and the outward appearance of a document at the same time. Its model of a document, therefore, must include both these aspects of a document. The typical text editor's model is inadequate for the Etude system because it includes no information about the document's outward appearance. The text formatter's model does include information about both the content and outward appearance of the document, but is also inadequate because it is not directly modifiable; a lengthy process is required to update the model whenever any change to the document is required.

The Bravo text editing and formatting system does represent the content and outward appearance of a document in a single structure that is directly modifiable. Etude, however, operates on an additional aspect of a document other than the content and outward appearance; this aspect is ignored by most text processing systems, including Bravo. We call this aspect the *internal structure.*

The internal structure is the organization and classification of the ideas and information contained in a document. In a report, the way the report is broken down into chapters and sections is part of its internal structure. The internal structure also includes other identifiable components of a document, such as quotations (excerpts), numbered lists, and italicized phrases. The internal structure even includes simple, commonplace components, such as paragraphs, sentences, and words.

The Etude system uses the internal structure of a document to determine its outward appearance.[3] In a typeset document, chapters are usually distinguished by starting a new page and using a larger type size for the chapter title. Quotations have extra space left around them, and a slightly smaller type size is used. Paragraphs have extra space left above and below them, and might have their first line indented. In this view, even sentences and words, seen as components of a document's internal structure, influence the outward appearance: some white space is left between words, and a little more space is left between sentences.

Typesetters have known for centuries that if a document's internal structure is clearly conveyed to the reader by its outward appearance, the document is easier to read and understand than if its outward appearance does not reflect its internal structure. When the outward appearance reflects the document's internal structure, the reader is given easily recognizable visual keys of the organization and kind of information in a document. Imagine reading a book with no noticeable breaks between chapters; with quotations that were run together with the regular text; with paragraphs that ran into one another, making it unclear where one ended and another began. The author of such a book (if he was a good writer) would know the

---

[3]The Scribe system [18] was the first system to recognize and use the internal structure of a document in the formatting process.

internal structure underlying his book, but unless the book's printer makes the reader aware of that structure by providing the appropriate visual keys, trying to read that book would be a nightmare.

The outward appearance of a well formatted document, therefore, is intimately tied to the information conveyed in the document. Different kinds of documents exist for different purposes, and convey different kinds of information. A *letter* is a printed message from a person or organization addressed to another person or organization. A *thesis* is a dissertation embodying the results of original research written by a candidate for an academic degree. A *wedding invitation* is a formal request for a person to be present at a marriage ceremony. Not only is the outward appearance of each of these documents different, but the internal structure is also dependent on the type of document.

A document's internal structure is partially determined by the document type. Each of the documents listed above has a different set of document components that make up its internal structure. A thesis, for example, has a title page, acknowledgements, a table of contents, a list of figures, a number of chapters and sections and appendixes, and a list of references. None of these components would appear in a wedding invitation.

A document's outward appearance is mainly determined by the document type and its particular internal stucture. Let us see how a letter's outward appearance is determined. A typical letter has a return address (or letterhead), a date, a recipient's address, a salutation, a body, a closing, and possibly some additional notations. Moreover, the body of a letter would itself have components, usually a set of paragraphs. Each component of a letter, and the letter itself, has an associated outward appearance. In a typical business letter, the return address appears in the upper right corner of the first page, and the date appears under the return address. Each paragraph in the body might be justified, with the first line indented, and with

some white space inserted between paragraphs.

The Etude system uses the internal structure of a document to compose the document's outward appearance. Etude's model of a document, therefore, must include information about the internal structure of the document, in addition to its content and outward appearance.

Now that the three aspects of a document relevant to the Etude system have been identified, a physical representation of this model, which will be manipulated by Etude, must be devised. The representation chosen will, of course, contain information about the content, internal structure, and outward appearance of a document. In addition, the representation of each of these three aspects must be easily modifiable, because the content and internal structure is changed during the editing process, and the outward appearance also changes as a consequence.

3.1 The Representation of a Document's Content

The content of a document consists of a sequence of characters. The entire sequence of characters is stored in a doubly-linked list structure, called the *text chain*. A text chain is a continuous, linear structure; there are no loops or breaks allowed in it. Each individual element of the text chain is called a *link*. A link contains a character, a pointer to the next element, and a pointer to the previous element.

With this representation one can easily get from one character to another, forward or backward, by following the appropriate pointer in the link. Characters may be inserted into the text chain by creating a link with the desired character in it, setting the previous and next pointers to the links before and after the insertion point, and setting the pointers of those links to point to the new link. To delete a character, the

last step is reversed; the pointers of the links surrounding the link to be deleted are changed so that they point to one another, and the link to be deleted is effectively removed from the chain. Each of these operations—moving forward and backward by characters, and inserting and deleting characters—are operations needed for editing the content of a document.

The representations of both the internal structure and outward appearance of a document, introduced in the following two sections, are woven over the content of the document. In order to maintain the relationship between these structures and the content of the document, we allow other objects to be inserted into the text chain. Thus, the text chain has links that do not contain characters, but contain objects that relate to the internal structure or outward appearance of the document.

3.2 The Representation of the Internal Structure of a Document

From an analysis of the internal structure of numerous kinds of documents, we have seen that the internal structure of most documents may be modeled as a hierarchy. For example, in a typical document, the characters are grouped into words, the words are grouped into sentences, and the sentences are grouped into paragraphs. In a sectioned document, the paragraphs are then grouped together into sections, and the sections into chapters.

The components of the internal structure of a document, except for words and sentences (whose representation is discussed below), are represented in a hierarchical tree structure. Each component is represented by an object called a *hlto*[4] (pronounced "hill-toe"). A hlto has the following information associated with it:

---

[4]Originally an acronym for "High Level Typographic Object."

- The hierarchical tree structure is implemented by each hlto having a single *owner* hlto and an array of *owned* hltos (children). The *root* hlto is the only hlto that has no owner; it contains all the characters in the document.

- Each hlto is of a specific *class*, which identifies the kind of component of the internal structure. Typical hlto classes are "report," "chapter," "quotation," and "paragraph."

- The hierarchical hlto structure is related to the content of the document: each hlto contains a region of text characters. Each hlto is related to the content of the document by having a *begin hlto marker link* and an *end hlto marker link* in the text chain. The begin hlto marker link is immediately before the first character contained in the hlto, and the end hlto marker link is immediately after the last character contained in the hlto. Thus, the characters contained in any hlto are accessible by using the begin and end hlto marker links to get to the text chain. The hlto structure is accessible from the text chain because each *hlto marker link*, either a *begin* or *end*, has a pointer to its associated hlto.

From any link in the text chain, the lowest hlto containing that link may be found by searching backwards through the text chain (towards the beginning) until a hlto marker link is found. If it is a begin hlto marker link, then the associated hlto is the lowest hlto containing that link. If it is an end hlto marker link, then the owner of the associated hlto is the lowest containing hlto. All hltos containing the link may be found by searching up through the hlto hierarchy, after the lowest containing hlto is found.

Just as the user, while creating and editing his document, changes the content by inserting and deleting characters, he also changes the internal structure, creating new hltos, and deleting existing ones. As he is typing a document, for example, he might be creating new "chapter" and "section" hltos. If he decides that he has started a new section unnecessarily, he needs to delete the "section" hlto.

More complicated operations, such as moving a paragraph from one point in the

document to another, require the insertion and deletion operations on both the content and internal structure. To move a paragraph, the characters contained within the paragraph hlto would be deleted, along with the paragraph hlto itself. Then the characters just deleted would be inserted at the new location, and a new paragraph hlto would be created around them and inserted into the hlto structure. This would, of course, be a simple "move paragraph" operation as far as the user was concerned; he would not be aware of the details of the implementation.

The implementation of those operations that perform hlto insertions and deletions is described below. The strict hierarchy of hltos is maintained through all insertions and deletions of hltos.

The following sequence of operations is performed to create a new hlto of a particular class in the document. The hlto is created around two links in the text chain; the pair of links delimit the characters contained within the hlto. (Actually, the two links need not be distinct, so that a hlto may be created around a single link.)

1. The links are checked against the existing hlto structure to make sure that the creation of a new hlto around those links would result in a valid hlto structure; this insures the hierarchical structure is maintained. In particular, a new hlto must be completely contained within an existing hlto; the new hlto cannot cross the boundaries of an existing hlto.

2. A new hlto is created, and its class is set.

3. The new hlto is inserted into the existing hierarchy. The new hlto's owner is set to the hlto immediately containing both links, and the new hlto is added to the owner's array of children. All the hltos in the array of owned hltos of the new hlto's owner are checked to determine if any are now children of the new hlto; those that are, are removed from the owner's array of children, placed in the new hlto's array of children, and their owner fields are updated to point to the new hlto.

4. A begin hlto marker link is inserted in the text chain before the first link

to be contained within the new hlto, and an end hlto marker link is inserted after the last link contained within the hlto. Both these links point back to the new hlto.

Any hlto except the root hlto may be deleted. To delete a hlto, the last two steps of the insertion operation are undone: the hlto marker links are deleted from the text chain and the hlto structure is updated. The hlto, removed from both the text and and the hlto structure, is no longer in the document.

As mentioned earlier in this chapter, words and sentences are not represented with hltos, although they are conceptually part of the internal structure of a document. There are two reasons for this:

- In a typical document, there are many words and sentences, while there are far fewer paragraphs, sections, and chapters. Words and sentences span only a few characters, while other components may contain a great deal of text. The representation of frequently occurring components should be compact for the sake of efficiency. Using the hlto structure, with its associated overhead, would be wasteful for words and sentences.

- Much of the reason for keeping an explicit model of the internal structure of a document is for its use in composing the outward appearance of the document. All the components of the hlto structure, in fact, have a specification of how that component should look (this is fully explained in Chapter 4). Words and sentences, however, only affect the outward appearance by having extra space left around them; the more flexible appearance specification associated with hltos is unnecessary for them.

For these reasons, an alternative representation was chosen for words and sentences. They are not represented with hltos, but are implicitly indicated in the text chain. In the text chain, there are links that demarcate words and sentences. These links contain either an *inter-word glue* or *inter-sentence glue*. (*Glue* is a kind of object that is like a character, but has some special properties. These properties are discussed in Section 3.3.1; the reader may think of glue as a blank space for

now.)

To find the word containing a character, the text chain is searched backward and forward for inter-word or inter-sentence glue (inter-sentence glue also indicates a word boundary); the characters between these two pieces of glue are the containing word. Similarly, a sentence is found by searching backward and forward for inter-sentence glue.

3.3 The Representation of the Outward Appearance of a Document

The outward appearance of a document is also modeled as a hierarchy. The model used in the Etude system is similar to the one used by TEX, a text formatter (see Section 2.5). Extensive use of TEX has shown that a hierarchical model of the outward appearance is adequate to represent the appearance of all documents.

In a simple document, characters are grouped together to form lines. The lines, in turn, are grouped to form columns, and columns are grouped into pages. A hierarchical structure may even be used to model the appearance of complicated document components, such as mathematical formulas; this is done by TEX.

The outward appearance hierarchy is built over the content of the document, like the internal structure hierarchy. Unlike the content or internal structure, however, the outward appearance is not built by the user. Instead, it is built automatically by the Etude system. Thus, in this chapter the outward appearance is discussed only as a static structure; how this structure is built and modified is discussed in Chapter 5, when we describe how the Etude system composes the outward appearance. The elements that the outward appearance hierarchy are constructed out of, *boxes* and *glue,* are discussed next.

3.3.1 Boxes and Glue

A *box* is the fundamental unit out of which the outward appearance of a document is built. Boxes are grouped together to form larger boxes, which are in turn grouped to form still larger boxes. A box is a two-dimensional object with rectangular shape. Boxes have a reference point, and three associated measurements, diagrammed below: [10]

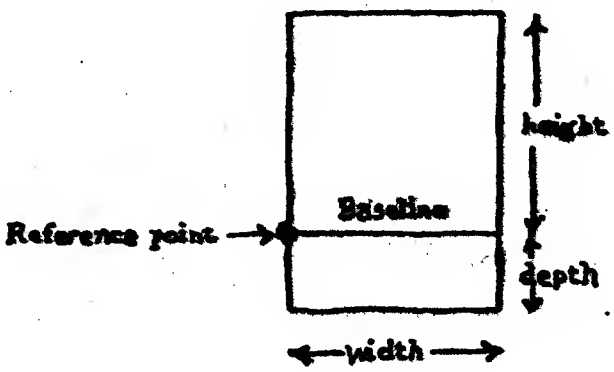

Figure 3-1: A Box and its Associated Measurements

When boxes are joined together to form larger boxes, they are either joined horizontally or vertically. If they are joined horizontally, they are all aligned on their reference points and the resulting box is called a *line.* The reference point of the line is the reference point of the first (left-most) box in the line. The height and depth of the line are the maximum height and depth of the component boxes. The width of the line is the sum of the widths of the component boxes.

A character is the simplest kind of box. Its reference point corresponds to the

base line of the character. The base line of a character is an imaginary line at the top of the descender of a character with a descender (for example, "g" or "p"), or the bottom of a character if it has no descender (for example, "a" or "b"). In typesetting, when characters are combined to form a line, they are generally aligned on their base lines. When constructing a *line* of characters, the base lines of the characters will be aligned properly, because the component boxes (characters) are all aligned on their reference points.

If boxes are joined vertically, they are also aligned on the reference points of the individual boxes, and the resulting box is called a *column*. The reference point of the column is the reference point of the last (lowest) box in the column. The width of the column is the width of the largest component box. For example, the typical column of text is a box constructed of lines. A line's reference point is normally at the left edge of the line, because the reference point of a line is the reference point of the first character. Thus, lines that are joined together to form a column are aligned on their left edges.

In addition to the three kinds of boxes already mentioned, there is another kind of box, called *glue*; glue is inserted into a box when extra space is needed between the component boxes. The details of the four different kinds of boxes—characters, glue, lines, and columns—are explained in the next few paragraphs.

A character has two components: its *identity* and the *font* to which it belongs. The identity simply indicates the letter of the alphabet that the character represents. A font contains all the characters of the alphabet in a particular type face and size. Associated with any font is a set of values for the height (distance above the base line), depth (distance below the base line), and width of each character in the font. This table of values is consulted whenever the box dimensions for any character are needed.

Glue is a kind of box used to represent blank space between the other kind of boxes on a page. In a line, glue has width but no height or depth, while in a column it has height but no width. The *class* of a piece of glue is analogous to a character's identity: it specifies the kind of glue. For example, the two classes of glue normally found in a line are *inter-word glue* and *inter-sentence glue*, and the glue normally found in a column is *inter-line glue*. (Inter-word glue and inter-sentence glue are also used to demarcate words and sentences in the text chain, as discussed in the previous section. Inter-sentence glue usually has a slightly larger width than inter-word glue.) Also like a character, a piece of glue has a font to which it belongs. (In addition, glue may also have a *mandatory line break* attribute, which signals the text formatter to break the line after the piece of glue.)

Glue has three attributes: a *natural space*, a *stretch*, and a *shrink*. A glue's class, combined with its font, determines its particular values of natural space, stretch, and shrink. The natural space is the normal width of the blank space in a line for a given font, or the normal height of the blank space in a column. The stretch and shrink components determine how much the normal blank space may be expanded or contracted if it is necessary to increase or decrease the blank space in order to, for example, justify a line of text.

Each piece of glue in a line or column box is assigned a specific width (in a line) or height (in a column) when the outward appearance is built by the Etude system (see Chapter 5). Assigning a specific measure to each piece of glue in a line or column is called *setting the glue*; this process is described in Section 5.2.

A line has a measure called its *natural width*, which is the sum of all the widths of the characters in the line and the natural spaces of all the pieces of glue in the line. If each piece of glue in the line is set to its natural space, then the width of the line is equal to its natural width. Glue in a line is set to its natural space when the line is not justified. If the line is to be justified, the width of the line may be different from

its natural width, so the right edge of the line is lined up with the right margin. To do this, the width of each piece of glue in the line may be enlarged or contracted when the glue is set, and the line's actual width may no longer equal its natural width.

A line also has a *shift amount*, which is used to position each line horizontally within its containing column. A line in a column may not have its left edge on the left edge of the column; its left edge may be somewhere to the right of the column's left edge. This happens, for example, when the first line of a paragraph is indented. The shift amount of the line is a measure that specifies the amount the line is to be shifted horizontally within the column.

3.3.2 The Outward Appearance Hierarchy

In a document, each character is a member of a single line and each line is a member of a single column. The columns are organized into pages, but this is done by a separate subsystem and is beyond the scope of this thesis. (For a brief description of how this is done and its relationship with the work described in this thesis, see Chapter 4.) This section describes the representation of lines and columns in a document.

All the characters in the document are contained in the text chain. The characters are grouped into lines with *line marker links*, which are inserted into the text chain. A line marker link indicates the start of a new line of text. Each line marker link has a *line* associated with it, and vice versa; the line includes all the characters between its line marker link and the next line marker link in the text chain.

The line containing any character in the document may be found by moving backward from that character through the text chain until a line marker link is encountered. The line containing that character is the line associated with that line

marker link.

Just as the characters in a document are stored in a chain and grouped into lines, the lines in a document are also stored in a chain and grouped into columns. The chain containing the lines in a document is called the *line chain*. The line chain may contain pieces of glue, in addition to lines. The class of glue found in the line chain is *inter-line glue*; these pieces of glue are used for leaving extra space between lines when necessary, such as the extra space between the last line of a paragraph and the first line of the next paragraph.

The line chain is broken into columns by *column marker links* in the line chain. Each column marker link signifies the start of a new column; it has a *column* associated with it, and vice versa. The lines contained in a column are all those lines between the column's column marker link and the next column marker link in the line chain.

Finally, the columns are themselves members of a chain, the *column chain*. The column chain, however, is only used for ordering the columns within the document, and is not used to represent the layout of columns on a page. As mentioned at the beginning of this subsection, the representation of the layout of pages is beyond the scope of this thesis.

3.4 Representing Changes to the Document

In previous sections, we have discussed how the content, internal structure, and outward appearance of a document are represented. We have noted that the user continually changes the content and internal structure, and we have shown how the representations of these two aspects are updated. We have not yet discussed how the outward appearance is composed, nor have shown how the outward appearance

This figure shows the *text chain, line chain,* and *hlto hierarchy* in a portion of a typical document. The links in the text chain are shown as little squares. They contain either characters, glue (empty squares), hlto markers (hm), or line markers (lm). The line chain, at the left, contains lines (line) and a column marker (cm). The entire text shown is contained in a "paragraph" hlto, and the word "the " is in an "italic" hlto.

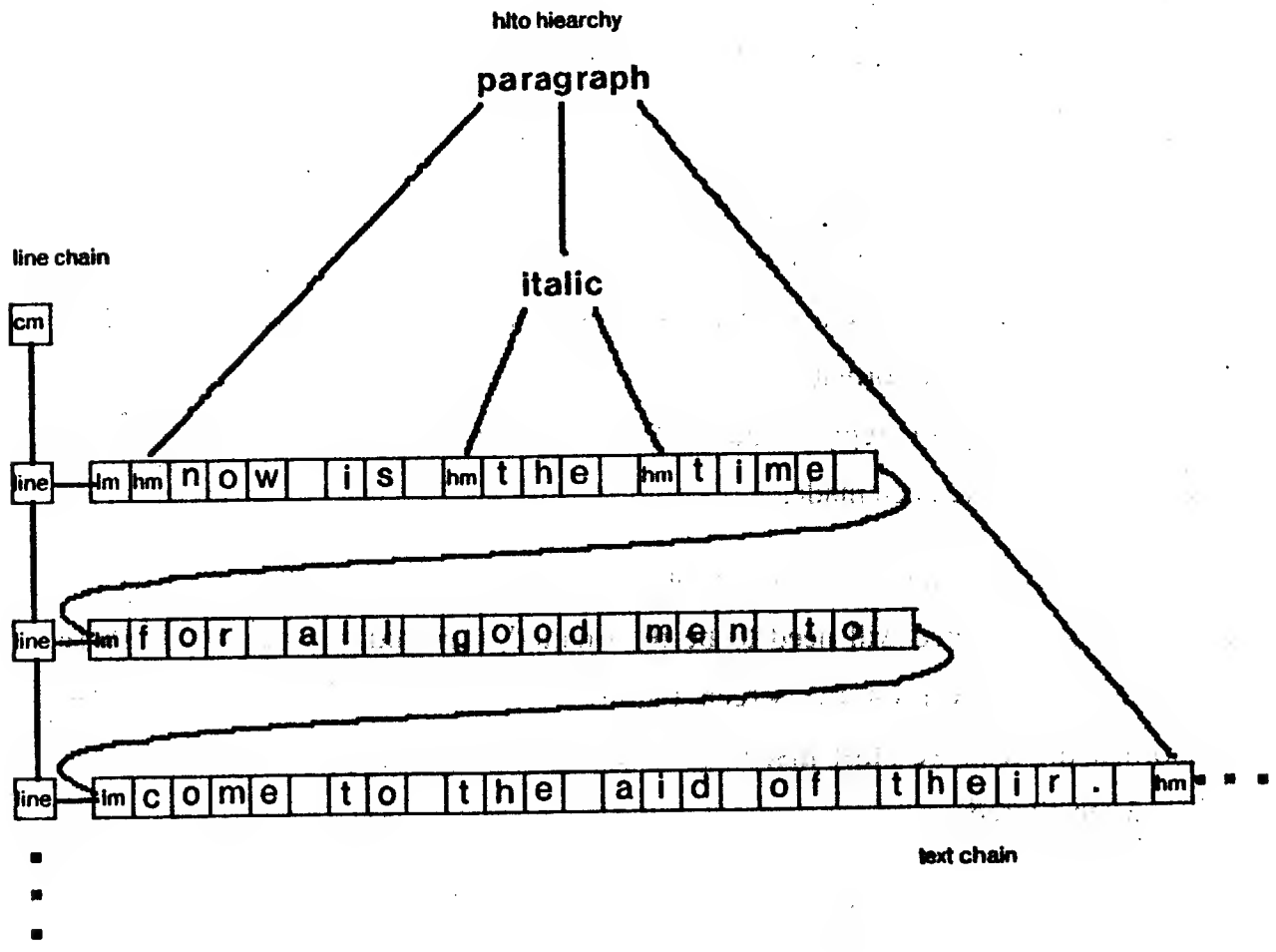

Figure 3-2: A Portion of the Content, Internal Stucture, and Outward Appearance of a Typical Document

is printed or displayed—these operations are detailed in Chapter 5.

In Chapter 1, however, we noted that the Etude system gives the user immediate feedback on the display of any changes he makes to the content or internal structure of his document. This cability requires the Etude system to be able to perform *incremental formatting* and *incremental redisplay*. Incremental formatting is the ability to format—i.e., reconstruct the outward appearance of—only those portions of the document that have been changed. Similarly, incremental redisplay is the ability to redisplay only those portions of the document that appear on the screen and have been changed.

In order to do incremental formatting and redisplay of the document as it is edited, the document must maintain indications of the changes that have been made to it. If this is done, the systems that format and display the document can interpret these indications, and format and display only the changed portions. Thus, we must make provisions in the representation of the document to indicate where changes to the document have been made.

How do we indicate what portions of the document have been altered? A section that has been altered will need to be reformatted, then redisplayed. (Only those sections that appear on the screen require reformatting.) But before we can reformat the text, we must first find—in an efficient manner—those sections that have been altered.

We might leave the indication of the change directly in the content of the document (in the text chain); this would require a search through all the text that appears on the screen. We could not quickly find the altered sections if we had to search through all the text, character by character. This suggests that a hierarchical representation of altered sections is desirable. With a hierarchical representation, we can quickly "zoom in" on those sections that have been changed, and ignore

large portions of text that have not been altered.

Rather than creating a new hierarchical structure to be used to represent altered sections, we should examine the existing hierarchical structures to see if these are adequate for the purpose. We could indicate altered sections of the document by leaving marks in the internal structure hierarchy or in the outward appearance hierarchy of the document. In fact, both methods were tried in different implementations of Etude.

If we use the internal structure hierarchy, which is represented with the hlto structure, then whenever a change is made in the text of a hlto, that hlto—and all hltos containing it—are marked as altered. A typical document is mainly composed of paragraph hltos at lowest level; thus, if a small change is made, an entire paragraph would be marked as changed. The entire paragraph would need to be reformatted and redisplayed. Our goal, however, is to reformat and redisplay as little as necessary, and in most cases we could do better than reformatting and redisplaying an entire paragraph.[5]

Instead, the outward appearance hierarchy is used to keep track of the changes made to the document. When a change is made to the document, the *lines* in the changed section are marked as changed, and the columns containing those lines are also marked as changed. With this scheme, the smallest unit of text that is reformatted and redisplayed is one line. Although this is not ideal—we might wish to redisplay only a single character—it is adequate for our purposes.

We have only discussed in a vague sense what it means for a section of the document to be "changed." We have said that such a section, if it appears on the

---

[5]The actual implementation had an indication of where the changed section began in the hlto. We therefore did not need to reformat and redisplay an entire hlto, but only the "rest of" the hlto, after the point where the change began. However, this is still too large a section of text.

screen, needs to be reformatted and redisplayed. Just marking a section of the document as "changed" is not adequate to fully represent the dynamics of formatting and display. For example, a section of the document that has not been changed may still need to be reformatted. This happens when a character is deleted from a line; the previous line has not been changed, but it still may need to be reformatted, because deleting the character may allow a word at the beginning of that line to move up to the end of the previous line.

Thus, there are actually two kinds of marking done on the document: *unformatted* marking and *changed* marking. Sections of the document that are *potentially* unformatted as a result of an editing operation are marked *unformatted*; this is an indication to the text formatter that it must examine that section and reformat it, if necessary. Sections of the document that have actually been altered are marked *changed*; this is an indication to the redisplay subsystem that these sections need to be redisplayed on the screen.

If a character is inserted or deleted from the text chain, then the line containing that character is marked as both *changed* and *unformatted.* As just mentioned, however, the previous line may need to be formatted, because some insertions or deletions cause a word to move to the previous line. This only happens if the insertion or deletion occurs before the first piece of glue in the line. (More precisely, it only may happen if the insertion or deletion occurs before the first character in the line at which the text formatter may break the line, and this can only happen at a piece of glue.) Thus, if the insertion or deletion occurs before the first piece of glue in the line, the previous line is also marked as *unformatted* (note it is *not* marked *changed*).

If a hlto is inserted or deleted from the hlto hierarchy, then all the lines that have characters contained in the hlto are marked as *changed* or *unformatted,* and the line before the first line is marked as *unformatted.*

This marking propogates upward through the outward appearance hierarchy. When a line is marked *unformatted* or *changed*, the column containing it is also marked *unformatted* or *changed*.

The text formatter formats a section of the document by formatting all the lines in that section that are marked *unformatted*. In doing the formatting, it may unformat and change additional lines of the document, and these lines are marked appropriately. When the formatter finishes formatting all the lines in the section, it marks those lines as *formatted*. The redisplay system, in order to keep the screen up-to-date, would then redisplay all the lines that were marked *changed*, and then mark them as *unchanged*. These procedures are described in detail in Chapter 5.

Chapter Four

Formatting Environments

The text formatter of the Etude system composes the outward appearance of the system's documents. Etude's formatter uses a data base of formats for determining how each component in the internal structure of a document should be formatted. It derives the formatting information from both the data base, which contains a set of pre-defined formats for each class of hlto, and the arrangement of hltos in the internal structure hierarchy.

The data base contains a *format specification* for each class of hlto known by Etude. The format specification includes a number of format *attributes*, and a *value specification* for each attribute. For example, "type face" is an attribute that might have the value specification "italic," and "right margin" is an attribute that might have the value specification "1.5 inches."

Usually, the attributes and value specifications only partially specify the formatting *environment* of a piece of text. The formatting environment is a total specification of all the typographic attributes and values in force at any point in the document. For those format specifications that do not completely specify the formatting environment, the desired value for the unspecified parameters may be derived from other format specifications. For example, a "center" hlto might be a document component that would center the text contained in it. We would want the text centered within the margins of the document, whatever they happened to be. The format specification associated with the "center" hlto would not specify the margins the text should be centered between; rather, the margins would be derived from the margins of the document type. Thus, the desired margins for the centered

text would be *inherited* from previous specifications.

The scope of formatting dealt with in this thesis is the formatting of the text of a document into columns, without regard to the placement of the text on pages. The *page makeup* subsystem in Etude [20] is responsible for defining the layout of a page. It "supervises" the text formatter by constraining the width of the lines that the text formatter composes. In traditional typesetting systems, the same separation of the two processes occurs: the text is formatted into galleys (of the appropriate width), and these galleys are cut and pasted together, usually manually, to form complete pages. In Etude, the "galleys" produced by the text formatter are cut and pasted together automatically by the page makeup subsystem.

A general description of the attributes in the environment for text formatting was given in Section 1.3. Before providing more details about these attributes and their allowable values, we first must look at the way measurements may be specified in Etude.

4.1 Distances

All measures in Etude are expressed in *distances*. Distances provide a uniform mechanism for expressing *absolute, environment-dependent,* and *device-dependent* measurements.

Absolute — An absolute distance specifies a measurement in terms of absolute units. The system recognizes any of the following common units for specifying absolute distances.

- inches
- centimeters
- millimeters

- points (a unit of 1/72 inch used by printers)

- picas (a unit of 1/6 inch used by printers)

Environment-Dependent

An environment-dependent distance specifies a measurement whose absolute value depends on the formatting environment in which the distance is evaluated. These are:

- characters

- lines

The "character" unit is the width of a typical character in the current font, which is determined by the environment. Similarly, the "line" unit is the height plus the depth of a typical line in the current font (which is the same as the height plus the depth of a typical character).

Device-Dependent

A device-dependent distance is expressed in units that are dependent on the resolution of the device that the document will be printed or displayed on. Because devices may have different horizontal and vertical resolutions, there are two device-dependent distances:

- horizontal units

- vertical units

Distances are used for two purposes: for the *specification* of measurements, and for the *evaluation* of measurements. Distances are usually specified in absolute or environment-dependent terms, and evaluated in device-dependent terms. This allows the format designer, who creates the data base of formats, to specify measurements, either absolute or relative, independent of any particular printing or display device. The formatter, which is composing the outward appearance structure for a particular output device, needs to evaluate these measurements in units specific to the output device. For example, a "quotation" might be defined to

leave 0.5 inches of space above and below it in the text; the distance would be defined in absolute terms, in inches. The text formatter would evaluate that distance specification in device-dependent terms; to do this, it needs to know how many vertical units that distance is on the intended output device.

4.2 Environment Attributes and their Values

The environment attributes, and values implemented, were chosen to provide a reasonably complete coverage of the requirements for formatting text into galleys.

As mentioned, the page makeup subsystem constrains the size (width) of each line formatted. Within the given size, the environment specifies a *left margin* and a *right margin* for the text, and an *indention* for the first line of the environment. The *left margin* and *right margin* attributes determine the *prevailing margins* of the environment. The values of each of these three attributes are distances.

Two attributes, *fill* and *justify*, determine how the text is broken and positioned within a line. *Fill* may take one of four possible values:

Fill	Instructs the formatter to include as many words as will fit on the line it is formatting. This is the normal value for formatting plain text.
Nofill	The formatter looks for pieces of glue having *mandatory line break* attribute, and breaks the line there. If no mandatory line break glue is found, the line is broken when it has become full. The text on the line is placed against the prevailing left margin.
Center	The same as nofill, except the text in the line is centered between the prevailing margins, rather than being placed against the left margin.
FlushRight	The same as nofill, except the text in the line is placed against the prevailing right margin.

Justify is a boolean value; it may either be *on* or *off*. If off, the glue in the line is set to its natural space. If on, and if the *fill* attribute has the value fill, then the glue in the line is set so the right edge of the line touches the prevailing right margin. (If the *fill* attribute has another value, justification is not done.)

The *type face* attribute specifies the type face in the environment. The value of this attribute may be *roman*, *bold*, or *italic*. These were the only three type faces available, due to the limitations of the display terminal used in the current implementation. If more type faces were available, additional values for this attribute would be accepted to allow selection of these additional faces.

The *leading* attribute, which takes a distance for a value, determines the spacing between lines in the text, in terms of the distance between the base lines of successive lines.

The *break* attribute determines whether a new line is begun on entering or leaving the environment. The possible values are: *before*, which makes the initial text in the environment begin a new line; *after*, which makes the final text in the environment end a line; *around*, which does both a break before and a break after; and *off*, which does neither a break before nor a break after.

Two attributes, *above* and *below*, are used for inserting extra white space between lines of different environments. *Above* specifies the amount of extra white space to be inserted before the environment, while *below* specifies the amount of extra white space to be inserted after the environment; both take distances as values. An above specification is ignored if the environment doesn't "break before"; similarly, below is ignored if the environment doesn't "break after." Of course, if the environment "breaks around," both above and below specifications are valid.

The remaining environment attributes all deal with *numbering*. If the *numbered* attribute, which takes a boolean value, is *on*, the hlto associated with the envi-

ronment is assigned a number, and this number is kept up-to-date automatically by the system. For example, "chapter" hltos are usually numbered; the system assigns a number to each chapter hlto, and updates these numbers when chapters are inserted or deleted from the document. The details of the numbering scheme are described in Chapter 6. A brief description of the other environment attributes pertaining to numbering follows.

If a hlto is numbered, Etude automatically prints the number in the style and location specified by the following two environment attributes. The *counter style* attribute allows the specification of a template, which determines the way the number appears in the document. The value of the counter style attribute is a string, but the string is interpreted in a special way. The hlto's number may be printed as an arabic ordinal, an alphabetic letter, a roman number, spelled out in words, or not printed at all. Any text may be printed along with the number. The *counter location* attribute specifies where the counter appears in the text. Two values for the counter location have been implemented: *flush left*, which prints the counter against the prevailing left margin; and *left flush right*, which prints the counter flush right against the prevailing left margin.

A hlto number may reference another hlto number when printing. For example, this section is numbered as section "4.2." In producing that number, the section references the number of the chapter that contains it. Such a reference, in this case the "4" referring to the chapter number, may be specified by the *counter style* attribute. If this is done, the *within* attribute, which takes a string as a value, specifies the hlto class within which the hlto is to be numbered. For example, the environment for section, to get the above numbering, has "chapter" as the value of the *within* attribute.

4.3 Format Specifications and Inheritance

In order to begin formatting at any point in the document, the text formatter must be able to determine the format environment at the point. There is a format environment associated with each hlto in the document. The format environment for the point is the format environment of the hlto containing the point.

The format environment of a hlto is derived from the class of the hlto, and all hltos above it in the hierarchy. The format data base contains *format specifications* for all the classes of hltos found in the document. These format specifications are used to derive the format environment of a hlto.

To determine the format environment for a hlto, we assume we have the format environment of its owner. The format specification associated with the hlto tells us how to change the format environment of the owner to get the format environment of the hlto. For example, the format specification of the "italic" hlto, used to change the type face of a region of text, would tell us to change the value of the *type face* attribute of the owner's environment, and assign it a value of *italic*; the other attributes of the format environment would be left unchanged.

A *format specification* differs from a *format environment* in two ways:

1. A format specification contains *value specifications*, rather than *values*, for each attribute in the specification. Depending on the particular attribute, a *value specification* for that attribute may contain a new value for the attribute, or may contain a way to derive the new value from containing environments. For example, the value specification for the *right margin* attribute might be "+1 inch"; indicating that the right margin for the new environment should be increased by one inch. The "+1 inch" value specification is not a *value* for the right margin; the actual value depends on the value of the right margin of the format environment of the containing hlto.

2. A format specification need not contain a complete set of value specifications for all environment attributes. There might be no value

specification for the *right margin* attribute in a format specification; the right margin would not change in the new environment derived from that specification. An "italic" hlto, used for changing only the type face of a region of text, would normally have no value specification for the right margin attribute.

Format specifications are stored in the data base of formats, which contains a set of format specifications for all classes of hltos known to the system. Each format specification for a particular class of hlto consists of a set of *attribute / value specification* pairs. The attributes are the same attributes detailed in the previous section. A value specification for many of the attributes is simply a new value for the attribute. For those attributes that take distances as values, however, a value specification is a distance, with an optional sign. The interpretation of the value specifications, particularly those with distances (signed or unsigned) is described below.

Left Margin and Right Margin

If the value specification is a signed distance, then it is interpreted relative to the value of the margin of the owner's environment. If the value specification is unsigned, then it is interpreted as a positive offset from the margins specified by the document type (the root hlto). If either or both of these attributes do not appear in the environment specification, then the values of the prevailing margins are used.

Leading

If the value specification is a signed distance, then it is interpreted relative to the value of the leading specified by the document type. If it the value specification is an unsigned distance, then that distance is the new value. If this attribute is not in the value specification, then the leading value of the owner's environment is used.

Fill, Justification, Type Face, Counter Style, Counter Location

The value specifications of any of these attributes, when they appear in an environment specification, are used directly as the new values for the attributes. For those that do not appear in the environment specification, the value in the owner's environment

is used.

Indent — If this attribute appears in an environment specification, then the associated value specification, whether a signed or unsigned distance, is used as the new value. If the attribute does not appear, then a value of *0* is used.

Break — If this attribute appears in an environment specification, then the associated value is used as the new value. If the attribute does not appear, a value of *off* is used.

Above and Below — If either of these attributes appear in an environment specification, then the associated value, whether a signed or unsigned distance, is used as the new value. If either attribute does not appear, a value of *0* is used.

Numbered — If this attribute appears and its value is *on*, then numbering is turned on in that environment. If it does not appear, then its value is *off*.

Within — If this attribute appears, then the hlto class name in its value specification is used.

In order to find the format environment for any point of the document, all the hltos containing that point are first determined. Beginning at the root hlto, this list is traversed, and format environments are successively generated for each hlto. When this is done for the last hlto—the hlto that immediately contains the point in the document—we have the format environment for the point in the document.

This method of generating environments is called an *inheritance* scheme because only a partial specification of the format environment is required for each hlto. Many of the parameters that are not specified at all *inherit* their values from the environments of hltos higher in the hierarchy. In addition, values for some attributes may only be specified relatively; in this case values from containing hltos are also inherited before the actual value for that format environment's attribute is determined.

Now that we have shown how the formatting environment for any point in the document may be determined, we proceed to explain, in the next chapter, how the text formatter works. The text formatter constructs the outward appearance of the document based on the format environments derived from the hlto hierarchy.

Chapter Five

Text Formatting and Display

As the user edits his document, the Etude system continually displays a formatted version of the document. After each editing operation performed by the user, Etude must reformat the document before it can be correctly displayed on the screen. The text formatter in Etude, described in this chapter, is responsible for reformatting the document. As mentioned throughout this thesis, the text formatter in Etude does as little formatting as is necessary to keep what the user sees on his screen correct; this is called *incremental formatting*. The display system in Etude, also described in this chapter, is responsible for maintaining an image on the screen of the outward appearance of the document. It also does as little redisplay as possible as the outward appearance changes; this is called *incremental redisplay*.

The Etude text formatter builds the structure that represents the outward appearance of the document. It does not disturb the content or the internal structure of the document. Two modules are involved in text formatting: the *dispatcher* and the *linewright*.[6] The dispatcher sequences through the lines of the document, invokes the linewright on those lines that may require formatting, and adds white space (glue) between lines as appropriate; the linewright sequences through the text chain and constructs lines based on the formatting environment (derived from the document's internal (hlto) structure). The linewright and the dispatcher can be invoked on any portion of a document, and will reformat the text

---

[6]Just as a "shipwright" builds and repairs ships, the "linewright" builds and repairs lines. The dispatcher module has been superseded by the *columnwright*, which, in addition to doing all the dispatcher does, builds and repairs columns in the same way the linewright constructs lines.

in that part of the document.

Most existing text formatters operate on an entire document at a time. In fact, they do not operate on a single document, but rather two representations of a document. The Scribe text formatter is typical of this:

> To use Scribe, you prepare a *manuscript file* using a text editor. You process this manuscript file through Scribe to generate a *document file*, which you then print on some convenient printing machine to get paper copy. [18]

In Scribe, the manuscript file contains the content and internal structure of the document, while the document file contains a representation of the outward appearance.

If a change is made to the manuscript file, the entire file must be run through Scribe for the change to be appear in the document file.[7] This requirement is inherent in the nature of existing formatters for the following reasons:

1. Editing is done using a completely independent text editing system. Because the editing and formatting activities are not integrated, the text formatter cannot determine what portion of text was altered.

2. There is no easy way to determine the formatting environment for the portion of text that has changed. Formatting commands at the beginning of the manuscript file may affect the formatting done at the end of the file. Thus, the only way to do derive the formatting environment at any point in the document would be to go through the entire manuscript file and accumulate all the formatting commands until that point is reached.

3. There is no formal connection between the manuscript and the document files. Even if the formatter could determine the place in the manuscript file that had been changed, and determine the formatting

---

[7] Scribe does have facilities that allow partitioning a large manuscript into a set of smaller manuscript files, each of which may be processed independently.

environment for that portion of text, there would be no simple way to update the corresponding document file.

The document representation of Etude is specifically designed to support incremental formatting. The capabilities that existing text formatters lack are found in Etude's document representation.

1. There is no division between the document representation on which editing is done and the one that represents the formatted document; in Etude they are the same. The editing primitives in Etude automatically mark the portions of the document that require reformatting.

2. The formatting environment at any point in the document can quickly be determined by using the hlto structure. All that is required is to search back through the text until the first hlto marker is encountered (see section 3.2); this is normally no more than a paragraph of text. The hltos higher in the hierarchy that contain that point are easily found by walking up the tree structure, and the inheritance mechanism can then be used to efficiently produce the formatting environment for that point.

3. Because of the single representation of the document employed by Etude, any changes in the document resulting from reformatting are reflected in the document without any additional work.

The following two sections describe in detail the operation of the dispatcher and the linewright.

5.1 The Dispatcher

The dispatcher is responsible for composing formatted galleys of text. It is invoked with a pair of line links in the line chain of the document, and it formats the text bounded by those two lines into a galley. The pair of lines on which the dispatcher is invoked would normally be the boundaries of the text that appears on the screen, which is all the text that need be formatted. The pair of lines might also

be the first and last lines of the document, which would result in the entire document being formatted into galleys; this would be done before the document was to printed.

The dispatcher sequences through the line chain, starting from the line link on which it was invoked. It checks each line it encounters to see if it is formatted; if it isn't, it invokes the linewright on that line. The linewright (described in the following section) formats the line and returns to the dispatcher. Note that the linewright may have unformatted succeeding lines in the document, but these will be formatted as the dispatcher sequences through the line chain. The linewright never unformats a line that preceeds the line it was called to format. Thus, when the dispatcher if finished, all the text between the pair of line links it was called with is formatted.

In addition to calling the linewright on unformatted lines, the dispatcher also examines the *desired space above* and *desired space below* values for every pair of adjacent lines, if one or both of the lines were unformatted. After both lines are formatted, it compares the *desired space below* the first line with the *desired space above* the second line, and records the larger of the two values. It then checks the links between the two line links to see if a piece of inter-line glue of the right size is there. It either updates the size of the glue, inserts a new piece of glue, or deletes the existing glue, as necessary.

5.2 The Linewright

The characters in the text chain of an Etude document are grouped into lines. Each time the document is edited, this grouping may become incorrect. For example, if characters are inserted into or deleted from the text chain, the existing line breaks in the text chain may not be correct; some lines may be too short or too

long. Also, if a hlto in inserted into or deleted from the internal structure, the margins might change, and the existing line breaks might again be wrong.

The basic function of the linewright is to examine the text chain of the document and determine, based on the formatting environment and the width of the column containing the text, how it should be broken into lines. As it scans through the text chain, it also performs some additional functions. The linewright is invoked on an unformatted line by the dispatcher, and does the following:

- It determines the formatting environment for the line and the width of the column containing the line.

- It sequences through the text chain from the start of the line and appropriately sets the type face of each character.

- As it sequences through the text chain, it determines where the line should break, and inserts a line marker link at this location (if one is not there already) to indicate the end of the line.

- It sets the glue and the shift amount in the line, based on the formatting environment.

- It sets the leading for the line.

- It leaves an indication of the desired space above and below the line. This is extra space over the normal leading (e.g., the extra space between paragraph). (There are two slots in each line for these values. When the linewright is done, the dispatcher examines these values and inserts the required space).

- It inserts into the line information necessary to create a *counter,* when necessary (see Chapter 6).

- It marks the line just formatted as *formatted.*

- If it has changed the location of the end of the line, then it marks the line as *changed.* In changing the location of the end of the line, it has also changed the contents of the next line of the document; therefore, it

marks the next line as *changed* and *unformatted.*

The remainder of this section is devoted to describing in detail how the linewright performs the tasks mentioned above.

The linewright is invoked upon an existing (unformatted) line of the document by the dispatcher. The linewright first determines the formatting environment at the beginning of the line. In addition, the linewright also gets the width of the column containing the line; this measure constrains the width of the line (see Chapter 4).

The links in the text chain are then examined individually, from the first link in the line (the link after the line marker link), sequencing forward through the text chain. As it examines each link, it maintains and updates various statistics about the line. These statistics include the line's natural width, stretch, shrink, height, and depth.

The *desired line width* is the size that the linewright attempts to make the line. The desired line width is initially set to the width of the column containing the line, less the sum of the left and right margins. The line is considered to be *full* when the sum of the natural width and the shrink of the accumulated links equals or exceeds the desired line width.

A decision is made to break the line if certain conditions are met at the time a link is examined. For example, the line is broken if a character is encountered and the line is full, or if a hlto marker link that calls for a line break is encountered.

Determining exactly where and when to break a line is not straightforward; in most instances when the linewright encounters a link that forces it to break the line, the linewright actually breaks the line at a different link, which may be before or after the link just encountered. The two common situations where this occurs are:

- When the linewright decides to break the line because the line is full, it must back up and break at the last inter-word glue.

- If there are several end hlto marker links in a row, each requiring a line break after it, the linewright should break only after the last one. Similarly, if there are several begin hlto marker links in a row requiring a line break before, the linewright should break only before the first.

Several strategies were tried for determining the exact point to break a line. The simplest and most successful strategy involves remembering and classifying *break points* in the line as the linewright scans through the text chain. The *break point* is the link that the linewright, as it scans through the text chain, has determined to be the best place to break the line so far.

There are four *break point classes*; they are, in order from lowest to highest: *none,* when no place to break the line has yet been seen; *possible,* when the linewright has encountered a link where it's possible to break the line (normally when the first character has been encountered); *desirable,* when the linewright has encountered an piece of glue (either inter-word or inter-line) at which it may break the line; and *necessary,* when a link that requires a line break has been encountered (such as an end hlto marker, whose corresponding hlto requires a ***break after***).

Each kind of link in the text chain has one of the above break point classes associated with it. As the linewright scans through the text chain, it compares the break point class of each link it encounters with the break point class of the line so far. If the link is in the same or higher break point class, then the link is remembered as the new break point, and the line's break point class is updated. In remembering a break point, the linewright not only remembers the link, but also records the natural width, stretch, shrink, height, and depth of the line at that point.

There are five different kinds of links that the linewright acts on when it encounters them in the text chain: characters, glue, begin hlto markers, end hlto

markers, and line markers. The actions that the linewright performs on encountering each of these links are detailed below.

Character

1. If the line is *full,* then the linewright ends its scan and breaks the line at the line's break point. If it is not *full,* then the line's break point class is checked and possibly updated:

   - If the line's break point class is *necessary,* then the linewright ends its scan and breaks the line at the line's break point.

   - If the line's break point class is *desirable,* then the line's break point and associated class are left unchanged.

   - If the line's break point class is *possible* or *none,* then the character is taken to be the new break point of the line. The new break point class of the line is *possible.*

2. The type face of the character is set appropriately.

3. The width of the character is added to the natural width of the line. If the character's height or depth is larger than the line's, then the corresponding measure of the line is increased.

Glue

1. If the line is *full,* then the linewright ends its scan and breaks the line at the line's break point. If it is not *full,* then the line's break point class is checked and possibly updated:

   - If the line's break point class is *necessary,* then the linewright ends its scan and breaks the line at the line's break point.

- If the line's break point class is *desirable, possible,* or *none,* then the piece of glue is taken to be the new break point of the line. The new break point class of the line is *desirable,* unless the glue has the *mandatory line break* attribute, in which case the new break point class is *necessary.*

2. The type face of the piece of glue is set appropriately. This is necessary because the inter-word glue and inter-sentence glue of different type faces may have different natural space, stretch, and shrink values.

3. The glue's natural space is added to the natural width of the line. If justification is on, the glue's stretch and shrink is also added to the stretch and shrink of the line.

Begin Hlto Marker

1. The new format environment is computed using the inheritance scheme, which uses the old format environment and the new hlto's class.

2. One of the following actions is taken, depending on the conditions:

   - If the new format environment is *break around* or *break before* and the break point class is *none,* then the desired line width measure is updated; the calculation is similar to the original one to determine the desired line width (the width of the column containing the line, less the sum of the left and right margins), except that the new format environment's indention is also subtracted from the containing column's width. The indention value is taken into account here because indention takes effect on the first line of the new environment. The desired space above the line, which is the value

of the format environment's *above* attribute, is recorded if larger than any *above* value encountered in the line so far. The break point class remains *none*. (The linewright also places a pointer to the hlto in the line; this pointer is used by Etude's numbering system; see Section 6.3.2.)

- If the new format environment is *break around* or *break before* and the break point class isn't *none*, then the linewright ends its scan and breaks the line at the link before the begin hlto marker.

- If the new format environment isn't *break around* or *break before*, then the desired line width measure is updated; the calculation is, as above, the width of the column containing the line, less the sum of the left and right margins, less the indention. The desired space above the line, which is the value of the format environment's *above* attribute, is recorded if larger than any *above* value encountered in the line so far. The break point class remains unchanged.

End Hlto Marker

1. If the new format environment is *break around* or *break after*, then the lines' break point is set to the end hlto marker link, and the line's break point class is set to *necessary*. The desired space below the line, which is the value of the format environment's *below* attribute, is recorded if larger than any *below* value encountered in the line so far. (If the new format environment is not *break around* or *break after*, only the following step is done.)

2. The new format environment is computed.

Line Marker The line marker link is added to a list of line marker links that

have been encountered in the line. The disposition of this list is described below.

At this point, when the linewright has ended its scan, it has determined exactly where the line should end; the last link included in the line is the break point. The linewright must now do the following things:

1. It must set the glue in the line. If the line is to be justified, then the glue in the line must be expanded or contracted so the right end of the line extends to the left margin. (If no justification is done, each piece of glue is simply set to its natural space.)

2. It must position the line horizontally in the column. It might position the line against the left margin, against the right margin, or centered between the two margins; it also may indent the line right or left away from the left margin.

3. It must set the height and depth of the line so that it is leaded (vertical line-to-line spacing) properly.

4. If the format environment requires extra space above or below the line, it must leave an indication of this in the line (it is the dispatcher's responsibility to actually insert this extra space in the line chain).

5. It must insert a line marker link after the last link in the line, and remove any old line marker links that should no longer be in the text chain.

Each of these operations is detailed in the remainder of this section.

How the linewright sets the glue in the line depends on whether it is trying to justify the line. If justification is off in the formatting environment, then all the glue in the line is set to its natural space. If justification is on, then the glue is set so that the width of the line equals the desired line width; this insures that when the left edge of the line is placed against the left margin (plus any indention), the right edge of the line will align exactly with the right margin. Actually, the linewright does not justify the line if the break point class is *necessary,* which means the linewright

broke the line before it was full. This occurs, for example, on the last line of a paragraph, which is normally not justified, because the glue on such a line might have to be stretched a ridiculously large amount.

When a line is to be justified, the glue in it is set so the actual width of the line equals the desired line width. First, the natural width of the line is compared to the desired line width to determine whether the glue in the line should be stretched (expanded) or shrunk (contracted). When setting the glue in a line that needs to be longer than its natural width, the extra space is distributed throughout all the pieces of glue in the line. It is actually distributed proportional to the amount of stretch of each glue; a piece of glue with a larger stretch gets more of the extra space than a piece of glue with a smaller stretch. Similarly, when the line needs to be shrunk, each piece of glue is shrunk proportional to its shrink.

Consider the following example of a line with four boxes separated by three pieces of glue: [10]

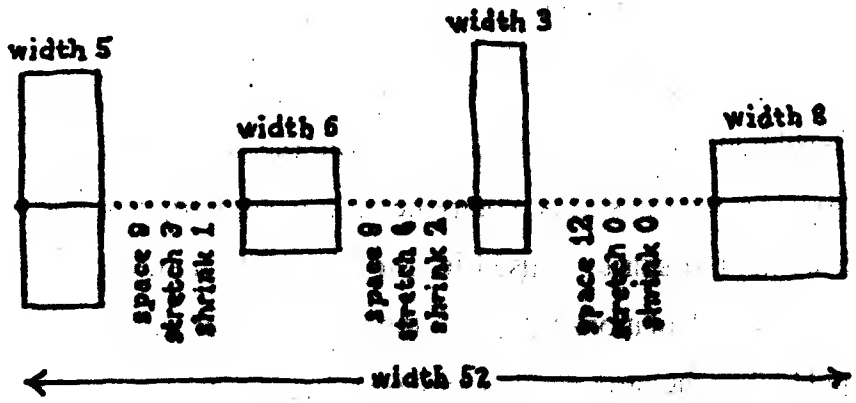

Figure 5-1: An Example of Setting Glue in a Line

The first piece of glue has 9 units of natural space, 3 units of stretch, and -1 unit of shrink; the next one has 9 units of natural space, 6 units of stretch, and -2 units of shrink; the last one has 12 units of space, 0 units of stretch, and 0 units of shrink.

The natural width of the line is 52 units, the sum of the natural space of all the glue and the width of all the characters. If we needed to make a line of 58 units, the difference between the natural width and the desired width would be 6 units; this is how much the glue would have to stretch. The stretch of the line is 9 units, the sum of the stretches of each glue in the line; the line's shrink is -3 units.

Let *glue.natural, glue.stretch, glue.shrink, line.natural, line.stretch,* and *line.shrink* be the natural space, stretch, and shrink of a piece of glue and the natural width, stretch, and shrink of the entire line. Let *line.desired* be the desired width of the line. Let *glue.width* be the actual width of a piece of glue after it is set in a line. Then each piece of glue is set according to the formula:

glue.width = *glue.natural*
+ (((*line.desired* – *line.natural*) * *glue.stretch*) / *line.stretch*)

In words, the total amount all the glue in the line must stretch is distributed over each piece of glue, in proportion to its stretch component.

In practice, each piece of glue in a line is set individually. After a piece of glue is set, it is considered to be of a fixed width in the remaining calculations. In particular, its stretch and shrink components are ignored, and its actual width (rather than its natural space) is used in calculating the natural width of the line. Thus, the following calculations to update the line's natural width and stretch values are performed before the next piece of glue is set:

line.natural = *line.natural* + *glue.width* – *glue.natural*

line.stretch = *line.stretch* – *glue.stretch*

The glue is set piece by piece, from left to right in the line. The width of the first

piece of glue would be set to:

= 9 + (((58 – 52) * 3) / 9) units
= 11 units

The next piece of glue would be set to:

= 9 + (((58 – 54) * 6) / 6) units
= 13 units

Since the last piece of glue has no stretch, its width would be set to its natural space, 12 units. The result of the glue setting operation is a line with a width of 58 units.

On the other hand, if the desired width of the line was 51 units, then we would have to shrink the glue in the line. In the case where the desired width of a line is less than its natural width, the following formula is used to set each piece of glue:

glue.width = *glue.natural*
+ (((*line.desired* – *line.natural*) * *glue.shrink*) / *line.shrink*)

After each piece of glue is set, the line's natural width and shrink are updated in the following way:

line.natural = *line.natural* + *glue.width* – *glue.natural*

line.shrink = *line.shrink* - *glue.shrink*

Using these formulas, we see that the first piece of glue is set to:

= 9 + (((51 – 52) * –1) / –3)
= 9 units

And the next piece of glue is set to:

= 9 + (((51 – 52) * –2) / –2)
= 8 units

The last piece of glue, because it has no shrink, is just set to its natural space.

After the glue setting, the line's width is 51 units.

After the glue in the line is set, the leading—the normal space between lines—of the line is set. The linewright leads the line by adjusting the height and depth of the line. If the leading attribute in the format environment of the line is larger than the height and depth of the line just created, then the linewright enlarges the line's height and depth to equal the leading value.[8]

After the leading of the line is set, the linewright then positions the line horizontally within the column by setting the shift amount in the line. The linewright sets the shift amount to position the line with respect to the left boundary of the containing column. It either sets the line flush left against the left margin, flush right against the right margin, or centered between the two margins, depending on the format environment. If the *fill* attribute in the reigning format environment has a value of either *fill* or *nofill,* then the shift amount is set to the value of the left margin, plus the indention value, if the line is the first line of a new environment. If the *fill* attribute has a value of *flushright,* then the shift amount is set to the desired line width less the line's actual width; this places the line flush against the right margin. If the *fill* attribute is has a value of *center,* then the shift amount is set to *half* the desired line width less the line's actual width, which centers the line between the left and right margins.

The linewright need only insert the desired space above and below the line into the line (there are two slots in a line for this). The dispatcher takes care of inserting the extra space in the line chain.

---

[8]This leading strategy is not quite correct. In particular, it can fail when type fonts of two different sizes appear on two successive lines. In these cases, pairs of adjacent lines must be examined and the distance between the base lines of these two lines should be set to equal the leading; this would been done by the dispatcher. The strategy described caused no problems, because all the fonts used in the current version of Etude are of the same size.

Finally, the linewright must insert a line marker link into the text chain (and a corresponding line link in the line chain) to indicate the end of the line. Before the linewright inserts a line marker link into the text chain, it checks to see if there is already one at the desired location; if there is a line marker link at the desired location, it need not insert one. If the linewright does insert a line marker link, it has changed the contents of some lines in doing so, and must mark the appropriate lines as changed. The details of this procedure are given below.

1. If there is a line marker link after the end of the line already, then the linewright does not insert a line marker link into the text chain. In this case, the linewright has not changed the contents of the line just formatted (probably; see step 3 below). Also, the linewright has not changed where the next line starts, so its contents have not been changed. Thus, the linewright need not mark either the current or the next line *changed.*

2. If there is no line marker link after the end of the line, then the linewright must insert one there (and a corresponding line link in the line chain). In doing so, it changes both the ending location of the line just formatted, and the starting location of the next line. To indicate this in the document, the linewright marks the line it just formatted as *changed,* and marks the next line—the line starting at the new line marker link—as *changed* and *unformatted.* (Because the starting point of the next line has changed, it must be reformatted.)

3. When a new line marker link has been inserted into the text chain, if necessary, any old line marker links that are between the start and end of the line must be removed from the text chain. As mentioned in the description of the linewright's actions as it scans the text chain, the linewright keeps a list of line marker links it has encountered. If there are any line marker links in the list, the linewright now removes each of them of them from the text chain, and also removes their corresponding line links from the line chain. If it removes an old line marker link from the line just formatted, the linewright marks that line as *changed.*

At this point, the linewright's job is complete. It marks the line it just completed as *formatted* and returns to the dispatcher.

5.3 The Display System

As the user edits his document, the Etude system continually displays a formatted version of the document. After each editing operation performed by the user, Etude formats the portion of the document that will be displayed on the screen. Etude then updates the screen to reflect any changes that have been made to the document.

In this section the general concepts behind the Etude display system are described. Only those aspects that relate to the interaction of display with editing and formatting are discussed; this includes the redisplay of *changed* lines, and the *block move* screen operation. The implementation of these operations is only described briefly, because the display system is not a major focus of this thesis. Other aspects of the display system—such as displaying on different devices, positioning "windows" on the screen, and positioning the document within a window—are not discussed here. For a complete discussion, see [19].

The display system translates the internal representation of the document into an image on a display terminal. Analogous to the text formatting operation, the goal of the display system is to minimize the amount of text that must be redisplayed after each operation on the document. Just as the objective of *incremental formatting* is to quickly format the minimum amount of text each time the document is changed, the display system does *incremental redisplay* to update the display efficiently.

Any line of text may have been marked *changed* by an editing operation, or by the text formatter. Such a marking indicates that the contents of the line have changed since it was last displayed. (See Section 3.4 and previous sections in this chapter for additional information on the *changed* attribute of a line.) If a line is changed (and it appears on the screen), it must be completely redisplayed.

Those lines that are not marked *changed* do not need to be redisplayed. Such

lines, however, may have *moved* to a different location on the screen. For example, if the user deletes a line of text in the document, then no lines below the deleted line are marked as *changed*. Nevertheless, all the lines below the deleted line need to be moved up on the screen. Thus, lines that are not marked as *changed* must still be examined to see if they have moved.

If a line or group of lines have not changed, but have moved on the screen, the display system uses the *block move* operation, rather than redisplaying the lines. The *block move* operation is an operation performed by the display terminal that moves text from one location on the screen to another. It is faster to use the block move operation than to send the text to the display terminal, in most cases.

The following paragraphs describe the implementation of incremental redisplay. First, we describe Etude's model of the contents (the image) of the screen. We then describe the process by which the screen, and Etude's model of the screen, is updated.

Etude maintains a model of the contents of the screen in terms of columns of the document that are displayed. Each column of text that is displayed on the screen is associated with a *column picture.*

A column picture contains information about the area of the screen in which it was last displayed. This information is used to implement the incremental redisplay and block move operation. The information is organized into a table of *screen records*, each associated with a line in the column. A screen record contains a pointer to a line, and the area of the screen in which the line was displayed. Because the information in column pictures is organized on a line-by-line basis, if any part of a line changes, the whole line must be redisplayed. Similarly, a block move always involves complete lines.

To update the screen after a change has been made to the document, Etude

invokes the *column picture print routine* on each column picture. The column picture print routine first checks to see if a block move within the column is possible. The routine looks through all the lines for a block of consecutive lines that are not changed, but are at a different position on the screen. (If more than one block is found, the block containing the most lines is chosen for the block move.)

The column print routine then instructs the display to perform the block move. It gives the display the area of the block to move (which it computes from the column picture's table), and the new location to which the block should be moved. The display itself takes care of all the details of moving the "bits" on the screen.

If a block move was done, the column picture print routine updates the column picture's table to reflect the new situation on the screen. Each of the old screen records falls into one of three categories:

Unchanged	A screen record that was completely contained within an unchanged area of the screen is not changed.
Moved	A screen record that was in the moved block simply has its area of the screen updated to the new value.
Changed	The other screen records (those that are replaced by the moved block) are removed from the table.

The last step in displaying a column picture involves displaying all the appropriate lines. A line is displayed only if it is marked changed or has moved on the screen. In order to display a line, the area on the screen that will be occupied by the line is first cleared, and then the text of the line is displayed (shifted horizontally by the *shift amount* of the line). Like the block move operation, the display handles the details of putting characters on the screen; the column picture print routine only gives the display the location of the first character in the line, followed by the sequence of characters in line. Glue is displayed by sending the amount of space to leave blank.

After a line is displayed, a new screen record is created for the line and put into the column picture's table. The line that was redisplayed is marked as *unchanged,* to indicate that it appears correctly on the screen. Screen records for lines that were not redisplayed (because they were neither changed nor moved) are simply left unchanged. If an old screen record points to a line that is no longer in the document—the line may have have been deleted from the document—then the screen record is removed from the table.

At this point, the display has been updated to reflect any changes in the outward appearance of the document, and Etude is ready to process another command from the user.

Chapter Six

Counters

Etude can automatically number components of the internal structure of a document (hltos), such as chapters, sections, and outlines. A hlto is numbered when the *numbered* attribute in its associated format environment is *on*. The hlto is assigned a *counter*, and the value of the counter is kept up-to-date automatically by the system. Hltos that are numbered are assigned numbers sequentially (within their owner hlto). For example, all the sections (a numbered hlto) within a chapter (the owner hlto), all the chapters in a document, or all the items in an outline, are numbered sequentially, starting with number "one."

If a hlto is numbered, Etude automatically generates and prints the number in the style and location specified by the following environment attributes. The *counter style* attribute allows the specification of a template, which determines the way the counter appears in the document. The hlto's counter may be printed as an arabic ordinal numeral, an alphabetic letter, a roman numeral, spelled out in words, or not printed at all. In addition, any text may be printed around the numeral. A hlto may be numbered *within* another hlto, so that the counter of a containing hlto may be included when the counter is printed; for example, the numbers printed for the sections in this thesis include the number of the containing chapter. The *counter location* attribute specifies where the counter appears in the line of text. It may appear flush left against the left margin, followed by the text of the line, the way the sections in this thesis are number. Or it may appear to the left of the left margin, as the list on page 84 is numbered.

The following sections describe the implementation of the automatic numbering

system of Etude. The representation of counters is described, followed by a discussion of how counters are kept up-to-date, and how they are printed.

6.1 The Representation of Counters

Automatic numbering is implemented with objects called *counters*. Each hlto that is numbered has an associated counter. A counter has several components:

Value — An integer that is the value of the counter.

Template — A sting that specifies the counter style (the outward appearance of the counter).

Value String — The actual text of the counter, derived from the value and the template.

Countee — The object being counted. It is a hlto in all cases in the current discussion.

Formatted and Changed flags — These are analogous to the *formatted* and *changed* flags associated with lines. They indicate whether the line the counter prints on needs to be formatted or redisplayed because of changes to the counter.

6.2 Keeping Counters Up-to-Date

Whenever the hlto structure is changed, that change may affect the existing counter structure. The operations that insert and delete hltos from the hlto hierarchy automatically invoke a procedure to update all the counters that are affected. The algorithm for updating the counter structure is conservative; it will generally update more counters than are necessary, but it is a simple algorithm, and works without problems in complicated situations.

Whenever a hlto is inserted or deleted, the *instantiate counter* procedure is called on the *owner* of the hlto that was just changed. Only hltos contained within the owner of the hlto just changed could have their counters affected. The *instantiate counter* procedure sequences through each child of the hlto with which it was invoked and updates the counter associated with each child.

In order to update the counter associated with each child hlto, the *instantiate counter* procedure first gets the formatting environment for the hlto. If the *numbered* attribute is *on*, then it assigns the correct value to the counter. The first child hlto of a particular class is assigned the value "1," the second child hlto of that class is assigned the value "2," and so on, for all the child hltos.

After the counter associated with each of the child hltos is updated, the *instantiate counter* procedure is called recursively on that child hlto, so all of *its* child hltos are updated. In this way, all the hltos that are potentially affected by a change to the hlto structure have their counters updated.

6.3 Formatting and Displaying Counters

Counters are associated with hltos, and are numbered with respect to the hltos' positions in the internal structure of the document. In order for them to appear when the document is displayed or printed, they must be instantiated into the outward appearance of the document. All components of the outward appearance of the document discussed so far have also been components of the content of the document. Counters are different: although they are part of the outward appearance, they are *not* part of the content of the document. The user does not type in the text of the counter, nor can he edit it directly (he would be able to change the way the counter appears by modifying the format specification of the hlto in the format data base). Because counters are not part of the text of a document, they are not

represented as elements of the text chain; they are represented in a special way as part of the outward appearance, as described in Section 6.3.2.

There are three steps to making counters appear on the display. First, the *value string* of the counter—derived from the counter's value and its template—is created. Second, the value string is instantiated into the outward appearance of the document in the appropriate position. Third, the display system redisplays the counter's value string whenever necessary, obeying the rules of incremental redisplay. The following three subsections describe how these tasks are realized.

6.3.1 Creating the Value String of a Counter

The *value string* of a counter is the text of the outward appearance of the counter. It is constructed from the counter's value and its template, whenever either is changed. The counter's value is just an integer; the counter's template is a string that is interpreted in a special way.

The value string is constructed from the counter's template and value according to the following rule. Any characters in the template that are not part of the set of specially interpreted character sequences (listed below) are copied directly into the value string. If any of the following character sequences appears in the template, then the value of the counter is converted to the corresponding textual representation of the value.

@1	Arabic cardinal numbers (1, 2, 3, ...).
@I	Roman numerals in capital letters (I, II, III, IV, ...).
@i	Roman numerals in lowercase letters (i, ii, iii, iv, ...).
@A	Capital alphabetic letters (A, B, C, ..., AA, AB, ...).
@a	Lowercase alphabetic letters (a, b, c, ..., aa, ab, ...).

@O	The name, with the first letter capitalized (One, Two, Three, ...).
@o	The name in lowercase (one, two, three, ...).

The value string for counters can depend on the position of the counter's associated hlto in the hlto hierarchy. An "outline," for example, has different numbering styles for the "items" contained within, depending on the nesting level of the "item." The first level of "items" may be numbered with capital roman numerals, the second level with capital letters, the third level with arabic numerals, and so on. Etude allows specification of such a numbering scheme in a template. Before we can explain how to do this in Etude, we need to explain how a counter's level of nesting is determined. First, we define a counter's *parent.*

A counter has a *parent* if the class of the hlto containing the counter's associated hlto matches the value of the *within* attribute of the counter's associated hlto, and the containing hlto is numbered. If the counter has a *parent,* then the parent is the containing hlto's counter.

The template for a counter might be divided in several sub-templates. This is used when the style a counter prints in depends on the its level of nesting; for example, the top level "items" in an outline are numbered with roman numbers, items contained within the top level items are numbered with capital letters, items within these are numbered with arabic numerals, and so on. Each of these styles is specified with a sub-template. If a counter has no parent (first-level nesting), then Etude uses the first sub-template to compose the value string. If a counter has a parent, but no "grandparent" (second-level nesting), then Etude uses the second sub-template. A template may contain an arbitrary number of sub-templates to support any number of nesting levels of counters. (If the counter is nested deeper than the number of sub-templates in its template, then the sub-templates are cycled through again. Thus, if d is the nesting level of a counter, and n is the number of

sub-templates in the counter's template, then the *i*th sub-template is used, where $i = d \bmod n$.) The following character sequence separates sub-templates within a template.

@,	Separates templates for different nesting levels of counters.

A counter may use the value string of a parent counter as part of its value string. For example, this subsection is numbered "6.3.1." The subsection counter uses the value of the containing section; the section, in turn, uses the value of the containing chapter. A particular character sequence in a counter's template is used to insert the value string of its parent counter in the counter's value string.

When a counter's value is referenced as the parent of another counter, we may want it to print differently than when it prints its value directly. For example, this chapter's number is printed in the form "Chapter Six"; that is, the text "Chapter " followed by the chapter number, spelled out. The sections in this chapter, however, are printed as "6.1" and "6.2"; the chapter number is referenced by the section number, but is printed as an arabic numeral. Etude allows the specification of two templates for each counter; one for when the counter is printed directly, and the other for when the counter is referenced as the parent of another counter. The following character sequences in a counter's template are used for printing parent counters as part of a counter's value string.

@#	Insert value of parent counter.
@|	Separates the template for a counter printed directly from the template for a counter printed as parent. The characters to the left of the @| are the template for when the counter is printed directly; the characters to the right of the @| are for when the counter is referenced as a parent. If there is no @| in the template, then the single template is used for both situations.
@:x	Insert character x if there is a parent counter.
@;x	Insert the character x if there is no parent counter.

The templates for the counters associated with the chapters and sections in this thesis are presented in the following table. These templates would be specified as the value of the *counter style* attribute for the "chapter" and "section" hltos.

Also, the value of the *counter style* attribute for an "outline" hlto is given. The "outline" hlto is not numbered itself, but it does have a value for its *counter style* attribute. This value is inherited by the "item" hltos contained within the "outline," which are numbered, and the *counter style* value becomes the template for these "items."

Hlto	**Counter's Template**
Chapter	Chapter @O@|@1
Section	@#@:.@1
Outline	@I. @,@A. @,@1. @,@a. @,@i.

6.3.2 Instantiating the Counter in the Document's Outward Appearance

The text of counters is instantiated in the outward appearance of the document during the text formatting process. By the time the text formatter is invoked on the document, the text of the counter has already been created, because it is updated whenever the value of the counter changes. The text of the counter is in the *value string* of the counter; what remains to be done is to instantiate the counter's value string in the correct position in the document's outward appearance.

The text formatter always places the value string of a counter on the first line of text of the associated hlto of the counter. It can put the value string in either of two locations on the line, as specified by the value of the *counter location* attribute of the counter.

When the linewright encounters a *begin hlto marker* whose associated hlto

requires a requires a line break before the hlto (the hlto's *break* attribute has a value of either *before* or *around*), it places a pointer to that hlto in the line (see Section 5.2). Each line has a slot for the linewright to put these hltos in. Since only hltos that require a line break before them can be numbered, any hlto that has a counter can be found in the line on which the counter will print. The document display system will check for these hltos in the lines it is displaying, and display the counter associated with each hlto, if any.

The linewright also positions the counter within the line. It does this by setting a value in the line for the *counter shift amount*. The *counter shift amount* is analogous to the regular *shift amount* of the line. The regular *shift amount* determines how much the text of the line is shifted from the left margin of the column in which it is printed; the *counter shift amount* determines how much the *value string* of the counter is shifted from the left margin.

If the value of the *counter location* attribute is *flush left*, then the text of the counter is placed at the beginning of the line, flush left against the prevailing left margin. The regular text of the line is positioned after the text of the counter. In order to get this effect, the linewright sets the *counter shift amount* to the value of the left margin of the prevailing format environment, and increases the regular *shift amount* by the *width* of the counter's value string. With these settings of the shift amounts, the counter's value string is printed at the left margin, followed by the text of the line.

If the value of the *counter location* attribute is *right flush left*, then the text of the counter is placed flush right against (and to the left of) the left margin. The linewright sets the *counter shift amount* to the value of the left margin less the *width* of the counter's value string; the regular *shift amount* is not affected. With these settings of the shift amounts, the counter's value string is printed to the left of the left margin, and the right end of the value string falls on the left margin; the text of

the line begins at the left margin.

6.3.3 Displaying Counters

The value strings of counters are displayed whenever the line they are on is redisplayed. This subsection outlines the modifications to the display system, as discussed in chapter 5.3.

A line is considered to be *unformatted* not only if it has been marked unformatted, but also if the counter on the line—if any—is marked unformatted. Similar, a line is considered to be *changed* whenever it, or the counter on the line—if any—is marked *changed.* And whenever a line is marked *formatted* or *unchanged,* any counter on the line is marked *formatted* or *unchanged.* Therefore, the formatter will reformat and the display system will redisplay any lines whose counter has changed, even if the text of the line has not changed.

In order to redisplay a line, the redisplay system first checks to see if there is a counter on the line. If there is, it first displays the counter, shifted horizontally by the *counter shift amount* of the line. Then the text of the line is displayed as before.

Chapter Seven

Evaluation and Extensions

In this final chapter we evaluate some of the design and implementation decisions of the pieces of the Etude system discussed in this thesis. Where appropriate, we suggest better ways to realize certain capabilities. We conclude with a discussion of possible extensions to the existing system, which lead into a discussion of an integrated office workstation.

7.1 The Document Representation

Etude models three aspects of a document: its *content,* its *internal structure,* and its *outward appearance*; this model has worked quite well for realizing the functions of Etude. In the current version of Etude, the content is represented with a link structure of character and glue links, the internal structure is represented with hltos, and the outward appearance is represented by a (limited) hierarchical boxes and glue structure, also represented with links.

The main problem with the representation of the content is storage inefficiency. The link structure simply uses too much space for storing each character. This limits the size of documents that can be manipulated by Etude. It also adversely affects the speed of Etude, making it difficult to realize the "immediate feedback" so important in making Etude easy to use. The hlto structure employed in Etude, which represents the document's internal structure, is basically adequate as it exists. The current box structure in Etude is only a partial implementation of a full, consistent, hierarchical box structure, similar to what is used by the TEX

system. [10] Additionally, we have found that even TEX's general model does not adequately address important *page layout* issues.

Instead of using the link structure to represent the content of a document, we could use an *array* structure. Arrays are more efficient for storing characters than links, because arrays do not have the overhead of *previous* and *next* pointers that links have. It is more difficult to insert and delete characters from an array structure, so the implementation of the editing operations would be more complex.

The existing version of Etude does not support a general hierarchical structure of outward appearance components: lines can contain only characters (and glue), while columns can contain only lines (and glue). Both lines and columns, however, are boxes, and should be able to be components of other lines and columns. The TEX system has this capability, and uses it to represent the appearance of complex mathematical formulas.

Representing the content of a document could be combined with representing its outward appearance, both using the same array structure. If we store the components of a line or a column in an array, and allow the components of the array to be characters, glue, lines, and columns, then we have a general hierarchical structure. Note that we would not need a separate structure to represent the content of the document; the characters making up the content could be obtained from the line arrays. Although it would be more difficult to "walk through" this hierarchical array structure to determine the content, such a representation both saves storage space and is completely general.

A new type of box, tentatively dubbed a *page box,* could be implemented for laying out pages. There are situations where horizontal and vertical lists of boxes (lines and columns) are not natural constructs for page layout. A page box would allow arbitrary positioning of component boxes within a larger box. For example, a

page in a document might have several columns of text, several cutouts for pictures, and a running header. A page box would allow these component boxes—columns (for the text), lines (for the header), and glue (for the cutouts)—to be directly positioned on the page, either absolutely or relative to other boxes. Although a structure with the same appearance could be build out of a general hierarchy of line and column boxes, it would be more cumbersome to do so, and much harder to manipulate if the page layout was altered. Although page boxes would mainly be used for page layout, they would also be primitive boxes, and would be able to be placed anywhere in the hierarchical box structure. If a page box was a good representation for a complicated piece of a mathematical formula or table, then it could also be used for them.

The line and column arrays must be able to have arbitrary objects inserted as components. We have this already to a limited extent in Etude's text chain, which allows hlto marker links (and other kinds of links) to be included in the chain; line and column arrays would also allow this. For example, a useful object to insert in a document would be a *cross-reference marker*. Like the Scribe system, Etude would insert the reference into the document and keep it up-to-date automatically. A *current date marker* would instruct Etude to insert the current date into the document (possibly offering a selection of styles).

An important feature lacking in Etude is the ability to *simulate the appearance of a document formatted for one device on another device.* The original design of Etude says that we should be able to "preview" on the screen how the document would look if printed. A *device-independent document representation* would provide this capability. With such a representation, any device driver routine would be able to interpret the document representation and print or display it, within the limitations inherent in the device. We have this to a limited extent in Etude, where a character in italics prints underlined on a display that doesn't have an italic type face. The

main problem in the existing implementation of Etude is that the the document is formatted for a device of a particular resolution, and thus cannot be printed on a device with a different resolution.

In a reimplementation of Etude, the document would be formatted for a particular device—the intended output device—and a simulation could be printed on another device—normally the display screen. This requires that:

- For each type face available on the output device, there will be a corresponding type face for each simulation device that approximates the appearance (the size and shape) of the type face of the output device. If an Etude document was normally printed on a particular electronic printer, we would have to create a type face for Etude's display for each type face available on the electronic printer. (This would be a difficult and time-consuming task because the simulation device—the display—would normally have a lower resolution than the output device.)

- Glue would not be set by assigning it a particular, device-dependent value, as is currently done. Rather, the formatter would specify the total amount the glue in the line (or column) needs to shrink or stretch. It would be the responsibility of the device driver routine to calculate a particular size value for each piece of glue. (This scheme would also reduce the storage requirements for glue, because a glue's size would not need to be stored for each individual piece of glue.)

With these changes we would be able to display a good approximation of the document's true appearance.

7.2 Formatting

We have evaluated and described improvements to the representation of the document; now we evaluate the formatting capabilities of Etude. The implementation of Etude generally meets the goal of providing most typographic capabilities for formatting *galleys* of text. Two basic capabilities are missing, though: the ability

to produce super- and sub-scripts, and the ability to use type faces of several sizes. The formatter cannot produce super- and sub-scripts because the document representation is not able to represent such a construct; once they could be represented (see the discussion above), the formatter could produce them. The formatter was built to handle multiple sizes of type, but this capability could not be tested because none of our display devices could display different sizes of type.[9]

The formatter could be improved in the way it breaks paragraphs into lines, and a hyphenation facility could be incorporated into the system. The TEX formatting system is sophisticated in these respects:

> When the end of a paragraph is encountered, TEX determines the "best" way to break it into lines. In this respect, TEX gives better results than most other typesetting systems [including Etude], which produce each separate line of output before beginning the next, because the *final* words of a TEX paragraph can influence how the lines at the *beginning* are broken. TEX's new approach to this problem . . . . requires only a little more computation than the traditional methods, and leads to significantly fewer cases in which words need to be hyphenated.

TEX's approach to paragraph breaking and hyphenation could be incorporated into the Etude system; the only potential problem is that we may not be able to satisfy Etude's requirement of real-time formatting with it, because it may require too much computation.

The Etude formatter should be extended to handle the layout of complete pages; we sketch here how this would be done. Formatting may be broken down into two activities, *text composition* and *pagination*. *Composition* is the process of setting the text of a document into galleys of type. *Pagination* is the art of page makeup, of arranging the columns of text and other document components, such as illustrations

---

[9] Actually, we were able to test this capability of the formatter to a limited extent, and it did work. The only problem was that lines were sometimes not leaded correctly, as explained in Section 5.2.

and folios, into finished pages. Pagination, as used here, subsumes and integrates three aspects of the overall document production process that have traditionally been handled as separate activities:

Pagination	The selection of page breaks, and the selection of folio style, placement and value for each page.
Makeup	The arrangement of composed type into pages, and the insertion of folios, running heads, and inserts in accordance with a designer's layout specifications.
Copyfitting	Determining the amount of space required to set a given amount of text, and the adjustments involved in making the text fit the space it is to be printed within.

The basic design problems to be handled include: design of an appropriate internal representation for a fully formatted document; design of a layout specification language to guide the system as it paginates a document; and the design of a set of algorithms to manipulate the internal document representation in order to achieve copyfit within the layout specification.

Other extensions associated with the formatting capabilities of Etude include: conversational formatting, in which the system prompts the operator to provide the components of the document being constructed; and an interactive analogic subsystem for creating or changing format definitions, with which the user could modify the format data base not by editing its text, but by indicating through examples the kinds of format structures desired.

7.3 An Integrated Office Workstation

A number of other document production facilities may also be integrated with Etude. These include an Etude extension for generating business graphics, such as tables and bar graphs. As in Etude, the operator would describe the desired

structure in high-level terms, providing a minimal amount of information; the system would then display a proposed candidate, which the operator will be able to modify. Other tools include a document analysis system, including a spelling checker and a syntax/punctuation/style checker; both of these would be integrated with the document processing system and would be available interactively. A reference and bibliography system could interface with an on-line bibliographical data base, providing an on-line search capability for this data base and a mechanism for automatic generation of appropriate references.

An integrated office workstation is more than just a collection of office tools. It must provide consistent user interfaces to the tools, uniform data structures underlying them, ready context switching among them, and a supporting infra-structure. We believe that there will be different versions of such a workstation for different classes of office personnel, such as clerical workers, professional, and managers. We also believe that a small set of fundamental capabilities form the underpinning of all the facilities to be provided in these various contexts. These include a text processing system (such as Etude), an office data base management system, an image handling system, and a communications mechanism. Out of this collection of tools can be built virtually any office application system. We will first have to identify the functions and facilities that these basic components should provide, and then design the common base of software that will underlie all of them. For example, an office data base system differs in a number of important ways from more conventional data base managers. It must be able to cope with multiple modes of data, including text, graphics, and images. It must deal with non-uniformly structured data, and must support very easy entry and retrieval of data. We view the office data base system as a universal filing system for all the documents and information bases used in the office, as well as a gateway into corporate data base systems. A variety of generic and specific office applications would be built on top of these basic building blocks. Among these would be an electronic mail system, a

forms handling system, a calendar manager, and a personnel tracker.

References

[1]

Anderson, Tim.
ETUDE Architecture.
Working Paper WP-022, Massachusetts Institute of Technology Laboratory for Computer Science, Office Automation Group, June, 1980.

[2]

Canning, Richard G.
Word Processing: Part 1.
EDP Analyzer 15(2), February, 1977.

[3]

Canning, Richard G.
Word Processing: Part 2.
EDP Analyzer 15(3), March, 1977.

[4]

Good, Michael.
Notes on the Etude User Interface Structure.
Working Paper WP-016, Massachusetts Institute of Technology Laboratory for Computer Science, Office Automation Group, October, 1979.

[5]

Good, Michael.
A Programmer's Guide to Etude.
Memo OAM-014, Massachusetts Institute of Technology Laboratory for Computer Science, Office Automation Group, April, 1980.

[6]

Good, Michael.
Etude and the Folklore of User Interface Design.
Memo OAM-018, Massachusetts Institute of Technology Laboratory for Computer Science, Office Automation Group, July, 1980.

[7]
Goodstein, David.
Output Alternatives.
Datamation 26(2):122-130, February, 1980.

[8]
Ilson, Richard.
An Interactive Editor and Formatter.
Working Paper WP-004, Massachusetts Institute of Technology Laboratory for Computer Science, Office Automation Group, June, 1979.

[9]
Ilson, Richard.
Recent research in text processing.
Words 9(1):32-34; 52-54, June-July, 1980.

[10]
Knuth, Donald E.
TEX and METAFONT: New Directions in Typesetting.
American Mathematical Society and Digital Press, 1979.

[11]
Liskov, Barbara, et al.
CLU Reference Manual.
Technical Report 225, Massachusetts Institute of Technology Laboratory for Computer Science, October, 1979.

[12]
Niamir, Bahram.
The Configuration of the Nu Terminal.
Working Paper WP-009, Massachusetts Institute of Technology Laboratory for Computer Science, Office Automation Group, July, 1979.

[13]
Niamir, Bahram.
The Editor System Display Device.
Memo OAM-009, Massachusetts Institute of Technology Laboratory for Computer Science, Office Automation Group, August, 1979.

[14]

Niamir, Bahram.
A Virtual Terminal Interface for Text Processing Applications.
Memo OAM-011, Massachusetts Institute of Technology Laboratory for Computer Science, Office Automation Group, December, 1979.

[15]

Office Automation Group.
Annual Progress Report.
Memo OAM-007, Massachusetts Institute of Technology Laboratory for Computer Science, Office Automation Group, June, 1979.

[16]

Office Automation Group.
Annual Progress Report.
Memo OAM-017, Massachusetts Institute of Technology Laboratory for Computer Science, Office Automation Group, June, 1980.

[17]

Pratt, V. R.
DOC Manual.
Massachusetts Institute of Technology, 1979.

[18]

Reid, Brian K. and Janet H. Walker.
Scribe Introductory User's Manual.
Second edition, 1979.

[19]

Rosenstein, Larry.
The ETUDE Redisplay Implementation.
Working Paper WP-021, Massachusetts Institute of Technology Laboratory for Computer Science, Office Automation Group, April, 1980.

[20]

Schoichet, Sandor.
Page Makeup in Etude.
Working Paper WP-020, Massachusetts Institute of Technology Laboratory for Computer Science, Office Automation Group, April, 1980.

[21]

Seybold, Jonathan.
Atex—Part I: The Atex-8000 as a Commercial System.
The Seybold Report 6(5), November, 1976.

[22]

Seybold, Patricia B.
Wang's 10A, 20 & 30 Word Processing Systems.
The Seybold Report on Word Proeessing 1(1), February, 1978.

[23]

Stallman, Richard M.
Emacs: The Extendible, Customizable, Self-Doeumenting, Display Editor.
Technical Report 519, Massachusetts Institute of Technology Artificial Intelligence Laboratory, August, 1979.

[24]

Alto User's Handbook.
Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto CA, 94304, 1979.